Introduction to Java Language

10

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Java 13, Java 11, and Java 9 improvements.
- Java concepts such as Java Virtual Machine, compiler, garbage collector, Java Runtime Environment, and Java Development Kit.
- Principles of object-oriented programming such as abstraction, inheritance, encapsulation, and polymorphism.
- How to install Java Development Kit.
- Importance of syntax and structure in Java.
- Reserved words, classes and objects, variables, methods, constructors, etc. using libraries.
- New features in Java 9 like Java Shell (RELP), Multi-Release JAR Files, Platform Support for Legacy Versions, Java Linker, Hash Algorithms, Modular Java Packaging, Tuning Improvements, G1GC, Process API Updates, and XML Catalogs.
- How to use the Integrated Development Environment such as Eclipse.

10.1 | Overview of Java

Java is one of the most popular languages used in the software world. It dominates most of the software – from household devices like microwave, washing machine, etc. to space technologies like NASA's Mars Rovers. Java works on "Write Once, Run Anywhere" (WORA) principle and hence can transcend many platforms. You may write a Java program on any device and compile into a standard bytecode. This bytecode can then run inside a Java Virtual Machine (JVM) installed on any other device, even if it is running a different operating system. This gives tremendous benefit as your program is now portable. There are endless possibilities of what you can do with Java.

Java 13 is the latest version of the famous programming platform released on 17th September 2019 which brings 5 JDK Enhancement Proposals (JEPs) and 76 new core library elements. Majority of these core library elements are just simple additions to the java.io package. Following are the Features you can expect in Java 13. The details can be found on the openjdk website (https://openjdk.java.net/projects/jdk/13/)

JEP 350 Dynamic CDS Archives: Java 10 brought an interesting change "JEP 310: Application Class-Data Sharing" which improves startup performance and footprint by helping JVM reuse class-data archive. This CDS feature allows application classes to be included in the shared archive. This JEP 350 enhances this feature further where at the end of Java application execution it allows dynamic archiving of classes. This archive includes all loaded application classes and library classes other than the ones present in the default base-layer CDS archive.

JEP 351 ZGC: Uncommit Unused Memory: This JEP is related to the garbage collector for claiming the unused heap memory by improving the z garbage collector.

JEP 353 Reimplement the Legacy Socket API: This JEP intends to modernize java.net.Socket and java.net.ServerSocket APIs by replacing the underline implementation which will provide easy adaptation of user-mode threads also known as fibers. These old APIs are from JDK 1.0 which uses PlainSocketImpl for SocketImpl. This has been replaced by NioSocketImpl in Java 13.

JEP 354 Switch Expressions (Preview): This enhancement brings a new keyword 'yield' to return a value from the switch. Previous to Java 12, switch was using a variable and then break statement to return a value. See the following example:

```
String result = "";
switch (character) {
    case a, b:
        result = "This is a or b";
        break;
    case z:
        result = "This is z";
        break;
    default:
        result = "This is other than a, b, or z. ";
        break;
};
return result;
```

In Java 12, we can simply use break to return a value.

```
String result = switch (character) {
    case a, b:
        break "This is a or b";
    case z:
        break "This is z";
    default:
        break "This is other than a, b, or z. ";
};
return result;
```

In Java 13, Java 12's break for returning value does not compile. Instead, we should use yield. See the following example:

```
return switch (character) {
    case a, b:
        yield "This is a or b";
    case z:
        yield "This is z";
    default:
        yield "This is other than a, b, or z. ";
};
```

However, the Java 12's introduction of new lambda-style syntax with the arrow is supported in Java 13. See the following example:

```
return switch (character) {
    case a, b -> "This is a or b";
    case z -> "This is z";
    default -> "This is other than a, b, or z. ";
};
```

JEP 355 Text Blocks (Preview): This JEP finally brings the most desired multi-line string literal, a text block that will allow to express strings which are placed on several lines. In the era of web development, many times it is essentials to send a multi-line data to the program but in the absence of such provision to accept, previous to Java 13 versions, we have to use multi-line as follows:

```
String myTextBlock = "<html>\n" +
                          " <body>\n" +
                                " <div>Hello World!!!</div>\n" +
                          " </body>\n" +
                     "</html>\n";
```

But with Java 13's text block, we can simplify this as follows:

```
String myTextBlock = """
                          <html>
                                  <body>
                                        <div>Hello World!!!</div>
                                  </body>
                          </html>
                     """;
```

With this you can see, how Java is constantly striving to be the best programming languages in the world by modernizing many of its features and providing greater support to cloud based applications.

Before Java 13, Java 11 was the version which caught the attention of many. The update (Java11) has brought some interesting features as well as removed some of the older technology features. Following is the list taken from the official site which shows the newly added and removed features (http://openjdk.java.net/projects/jdk/11/).

1. JEP 181: Nest-Based Access Control
2. JEP 309: Dynamic Class-File Constants
3. JEP 315: Improve Aarch64 Intrinsics
4. JEP 318: Epsilon: A No-Op Garbage Collector
5. JEP 320: Remove the Java EE and CORBA Modules
6. JEP 321: HTTP Client (Standard)
7. JEP 323: Local-Variable Syntax for Lambda Parameters
8. JEP 324: Key Agreement with Curve25519 and Curve448

9. JEP 327: Unicode 10
10. JEP 328: Flight Recorder
11. JEP 329: ChaCha20 and Poly1305 Cryptographic Algorithms
12. JEP 330: Launch Single-File Source-Code Programs
13. JEP 331: Low-Overhead Heap Profiling
14. JEP 332: Transport Layer Security (TLS) 1.3
15. JEP 333: ZGC: A Scalable Low-Latency Garbage Collector(Experimental)
16. JEP 335: Deprecate the Nashorn JavaScript Engine
17. JEP 336: Deprecate the Pack200 Tools and API

Before this update, Java 9 brought some of the major changes. It was finalized in 2011 and released in September 2017. Oracle described that they wanted to bring out a version that had a better default garbage collector, ability to deal with large heap sizes, and an improved JVM with a self-tuning capability. The elements of Project Jigsaw are also implemented in this version, which calls for a modular design for quick improvements and future enhancements. The final version that came out late September 2017 has most of the issues ironed out, and now it is time to start using the powers of Java 9.

There are several enhancement protocols such as JDK Enhancement Proposals (JEP) which are implemented in the Java 9 version. Some notable ones are the following:

1. **JEP 193:** It introduces the use of variable handles with the ability to create equivalent functionality in a variety of situations.
2. **JEP 222:** It implements a fully functional Java Shell which allows the use of a command prompt to run single line functions and statements.
3. **JEP 266:** Java 9 is big on concurrency and brings in the elements of reactive programming. This is introduced using a flow class with the capability of allowing the use of individual threads.
4. **JEP 268:** It introduces the use of XML Catalogs that allow the use of other languages to be implemented and integrated with the Java programming efforts.
5. **JEP 282:** It introduces the linking tool creator in Java. It can combine different modules and use the available dependencies to create customized processing solutions. It produces an executable file which includes the JVM structure required to run the Java program in any environment.
6. **JEP 295:** It describes the use of Ahead of Time (AOT) compilation, which reduces the load on the hosting processors and offers significant advantages in a variety of settings.
7. **JSR 376:** It describes the modularization concept which is introduced as the latest update through Project Jigsaw creating the platform module system for producing enhanced functionality.

10.2 | Basic Java Concepts

Java is one of the most widely used programming languages. It is an object-oriented language that uses a system of class hierarchies to create and manage the required functionality. The main advantage of Java is that it is designed to run on any system with the support of a virtual machine, in the form of JVM. The virtual machine creates a desirable environment for a Java program on all physical systems. It works by creating a bytecode that can be simply executed in an installed JVM environment.

With the latest updates, Java further enhances the main capacity of the language to offer a portable solution, where there is no need to recompile the program. The main syntax of the language is driven from C, especially the C++ platform. However, it is different because it focuses on creating solutions, rather than offering low-level functions, which are a trademark of the C language.

Now, we will discuss a few important concepts that you need to understand before starting on writing programs in Java.

In the Java world, you will frequently come across with words like JDK, JRE, and JVM as these are the core-essence of Java programming language. As a programmer, you may never need to go in depth of these concepts as you will be just using them to configure in Integrated Development Environment (IDE), but it is good to understand them properly in order to use Java to its fullest potential. So, let us first understand each concept and then see how they differ. Figure 10.1 shows how these components are attached together.

Figure 10.1 Components of Java language.

10.2.1 Java Virtual Machine

Java Virtual Machine (JVM) is the name given to an abstract computing structure that allows a computer with any specification to run a Java-based program. Each JVM is uniquely defined by three concepts:

1. *Specification* that documents how the JVM needs to be implemented through the computing resources. A single specification will suffice for all types of implementations.
2. *Implementation* is the next notion. It is a computer program that codes the requirements presented by the JVM specification.
3. The last part is an *instance*. This is simply the implementation program working on an available process to execute the Java bytecode generated by Java compilers.

JVM is what makes Java platform independent. Every OS has different JVM. However, the output generated after the bytecode is the same across all OS.

10.2.2 Compiler

Java code can be written in any text-based application. However, it only becomes a source code when you save a program on a relevant platform in the form of a .java file. This file needs to be compiled in order to become a program, which can be used as an executable version. The Java compiler is responsible for checking the code for compliance with the language rules, such as following the defined syntax.

Once source code is found to be compatible and without errors, the compiler converts the program into bytecode which results in a *.class* file. This file contains the instructions which are required for running the program on any JVM. The compiler is different because it does not produce the machine code for a particular CPU; instead, it generates instructions which can be run on any device with the installed JVM or Java Runtime Environment (JRE).

10.2.3 Java Runtime Environment

Java Runtime Environment (JRE) is a software suite which is installed on physical machines to enable the execution of Java programs. It is a package that combines a JVM with the class libraries that are required to run any defined bytecode in Java. Oracle provides its own JRE as a package in the form of HotSpot – their proprietary software package.

10.2.4 Java Development Kit

Java Development Kit (JDK) is a complete collection of all the tools that Oracle provides for Java. It provides a combination of programming, compiling, and execution elements which are required by Java programmers. You can use the OpenJDK environment, which is an open-source solution offered by Oracle. There are several companies that also offer their development kits since JDKs are freely available for use.

Programmers need access to a JDK, which may include a graphical user interface (GUI) as well to help implement convenience in programming activities. We especially look at Eclipse in this guide to carry out the ideal Java-based projects.

Flash Quiz Which one will you choose to install on your development machine – JDK or JRE?

Please follow Table 10.1 to understand the differences between JDK and JRE, so you will know which one to use and when.

Table 10.1 Differences between JDK and JRE

JDK	JRE
Development kit	Implementation of JVM
Useful for compiling Java program	Useful for running Java program
Contains Compiler and Debugger	Does not contain Compiler and Debugger
Needs more disk space	Smaller than JDK

Figure 10.2 shows how your Java program gets executed through JDK, JVM, and JRE.

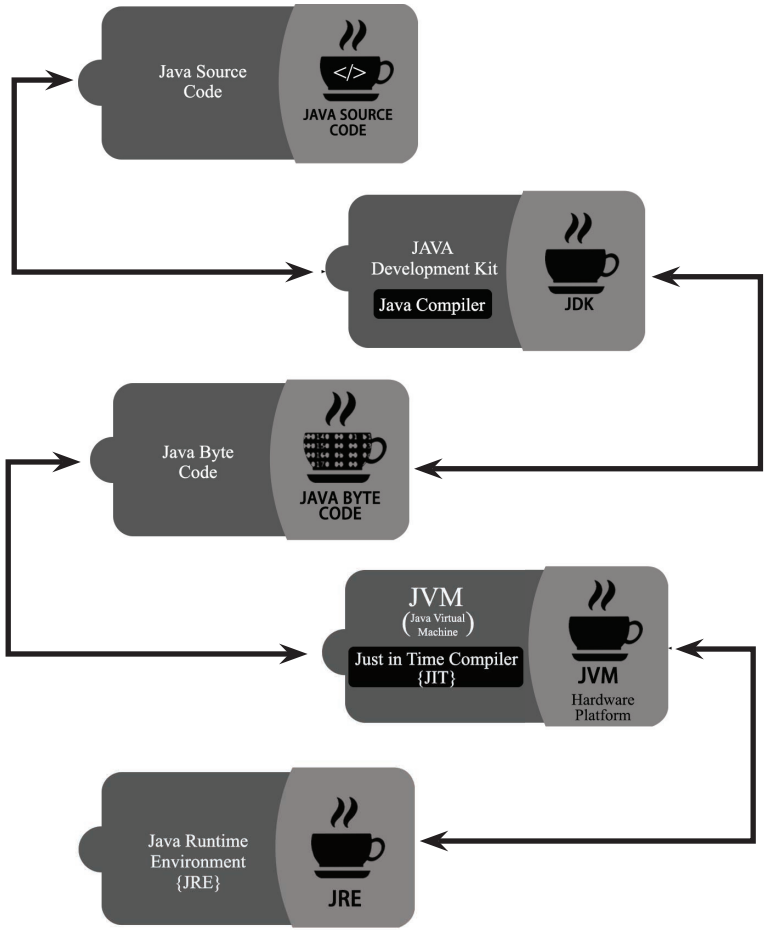

Figure 10.2 Java program execution flow through JDK, JVM and JRE.

10.2.5 Garbage Collector

One important benefit of Java over other languages is the unique way in which it addresses memory allocation and use. In fact, one of the most notable improvements in Java 9 is to specifically improve the memory handling, termed as garbage collection (GC). The JVM creates a memory space for each object from a defined memory heap.

Since a pool of storage is already available, a program in Java can quickly start on any device with a JVM. However, a garbage collector always runs in the background to keep a tab on how the available memory is employed. It checks the use of objects and reclaims the memory from objects which are no longer required by the Java application.

Java provides implicit memory management, which eliminates the need to implement any code that relates to performing the required memory management within the program. Memory management is a fundamental Java feature, which gives the program an edge when using programs that can be scaled on different levels to offer the same level of service and functionality.

Figure 10.3 shows how garbage collector helps in cleaning the orphan object references. We will cover this concept in more detail in Chapter 17.

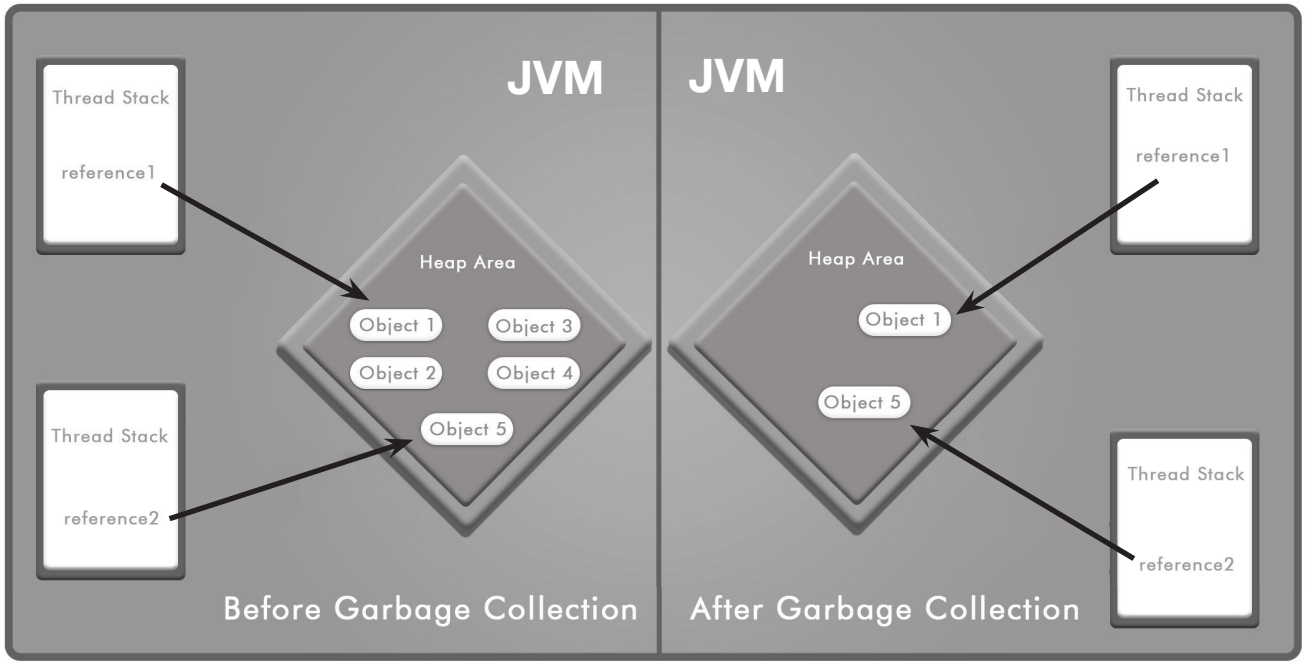

Figure 10.3 Objects state after execution of garbage collection.

Flash Quiz

If a variable is accessible by a thread, will it get collected by the garbage collector?

10.2.6 Objects in Java

Java is an object-oriented programming language and is different from its parent language C from which it borrows its main syntax and structure. It is a language which combines operational instructions with the required data elements into what we term as objects. These objects are instances that contain the complete structure which describes both the attributes of the functionality as well as how it will behave in all environments.

Objects allow the use of program logic in an integrated manner, and therefore make Java ideal to implement high-level logic and programming potential. The objects are used for all kinds of functions in Java. Objects often consist of subclasses like parent or child objects.

There are various important concepts about objects that we will discuss when describing the program syntax and how to write the Java code.

10.3 | Principles of Object-Oriented Programming in Java

As discussed earlier, Java is an object-oriented programming language (OOP). Every Java program follows these important OOP principles. The OOP principles are designed to combine the attributes of a function, with the required behavior in a single package, which is termed as an object. In the following subsections we will define four important OOP principles that you will observe throughout each course and tutorial available on Java programming – Abstraction, Encapsulation, Inheritance, and Polymorphism.

Before we begin with the principles lets first understand two important concepts IS-A and HAS-A relationship.

10.3.1 IS-A and HAS-A Relationships

IS-A and HAS-A relationships define how different classes are related to each other. Let us explore these concepts in detail below.

10.3.1.1 IS-A Relationship

IS-A relationship exists when one object is type of a particular class. In other words, the given object IS-A class. For example, Cat is a type of Animal so you can say that Cat is an Animal. Cat got all the properties of Animal and a little more properties of its own. Another example would be, Toyota Corolla is a type of Car so it is safe to say that Toyota Corolla is a car. Let us see the following example which shows this relationship.

```
package java11.fundamentals.chapter10;
public class Vehicle {
}
```

In the above example, Vehicle is a class which contains the information of Vehicle.

```
package java11.fundamentals.chapter10;
public class Car extends Vehicle{
}
```

The above code shows a Car class extends Vehicle class that means Vehicle is a superclass and Car is a subclass. Hence, as per our IS-A definition, Car is a Vehicle. Now, there can be many types of Cars which will have all the things that Car class has. So let us create a ToyotaCorolla class which extends Car class.

```
package java11.fundamentals.chapter10;
public class ToyotaCorolla extends Car {
}
```

Hence, as per our definition, we can safely say that ToyotaCorolla is a Car. Now, let us look into the definition of these classes in terms of Object Oriented terms.

In object-oriented terms, the class relationships can be defined as follows:

1. Vehicle is the superclass of Car.
2. Car is the subclass of Vehicle.
3. Car is the superclass of Toyota Corolla.
4. Toyota Corolla is the subclass of Vehicle.
5. Car inherits from Vehicle.
6. Toyota Corolla inherits from both Vehicle and Car.
7. Toyota Corolla is derived from Car.
8. Car is derived from Vehicle.

9. Toyota Corolla is derived from Vehicle.
10. Toyota Corolla is a subtype of both Vehicle and Car.

According to IS-A relationship, the following statements are applicable:

1. "Car extends Vehicle" means "Car IS-A Vehicle".
2. "Toyota Corolla extends Car" means "Toyota Corolla IS-A Car".

10.3.1.2 HAS-A Relationship

As we have seen in Section 10.3.1.1, IS-A relationship is based on inheritance. IS-A relationship is based mainly on how one class is related to the other. HAS-A relationship is different as it is about usage and not inheritance. In other words, how one class is used in another class. Let us see the following example to understand this better.

```
package java11.fundamentals.chapter10;
public class ToyotaCorolla extends Car {
    public static void main(String args[]) {
        NavigationSystem ns = new NavigationSystem();
        ns.getDirectionInstructions("New Delhi", "Agra");
    }
}
```

The above class ToyotaCorolla uses another class named "NavigationSystem" and then calling its method "getDirectionInstructions". See the following NavigationSystem class.

```
package java11.fundamentals.chapter10;
public class NavigationSystem {
    public String getDirectionInstructions(String from, String to) {
        return "Directions from " + from + " to " + to;
    }
}
```

In the above case, ToyotaCorolla and NavigationSystem are not related by inheritance but by usage. NavigationSystem object is used in ToyotoCorolla object to get direction instructions from one place to another.

10.3.2 Abstraction

Abstraction is the idea of revealing what is needed and hiding data that was not intended for the outside world. A simple example would be a car that is presented as a whole unit and not in the form of components. A user interacts with the car and accesses the required components like driving wheel, gears, etc. and not engine and other core components.

Java uses objects as abstract structures that provide usefulness in a variety of situations. Programmers can create all kinds of objects – individual variables, applicable functions, or data structures – for use in different parts of the same program. It is also possible to create classes of objects. This allows specifying the method of an object with the available syntax in Java.

Abstraction is possible because you can create specific classes of variables and datasets by setting up objects. Take the example of using a class that defines addresses. You can set up parameters such as the name of the person, the zip code, and the postal address within the class objects. In turn, different object instances may describe different addresses for a business, such as an employee and customer addresses.

Another real-life example would be an Automated Teller Machine (ATM) which dispenses money, tells us account balance, allow us to change card pin, etc. but we do not know the inner working of the ATM for performing all these operations for us (Figure 10.4).

Figure 10.4 Example of Abstraction.

10.3.3 Encapsulation

All objects are discrete, which means that all objects carry all the necessary information in their structure to execute successfully. This concept is termed as an encapsulation. The details and specifications of the objects remain hidden on the outside; this gives them improved protection which is not available with other programming languages.

The objects in Java can have relationships with each other, but they always have a boundary layer that isolates them from the external language environment. There are access modifiers available in Java which provide greater control over the objects, turning their status from public to private. A private object is the one whose attributes can only be read from within the object.

Java programming offers excellent encapsulation control as you can always change the width of the boundary of individual objects. This allows Java programming to offer superior solutions that have protected elements, which can be easily used in different elements thus maintaining their internal structure of attributes.

You can always reuse any function while ensuring the highest security. This is important because it saves time, while still ensuring a private setting. Once a specific code is created in the form of an object, it can be used with any database or process, while still ensuring that the actual information available in the database remains private within each object instance.

The availability of encapsulation allows programming to keep the original code secure, while others to use it for the intended functionality. This is excellent when collaborating with others, which is another strong point in Java programming. The Oracle Community allows you to share your problems and codes with other programmers to implement improvements and new library elements. Refer to Figure 10.5.

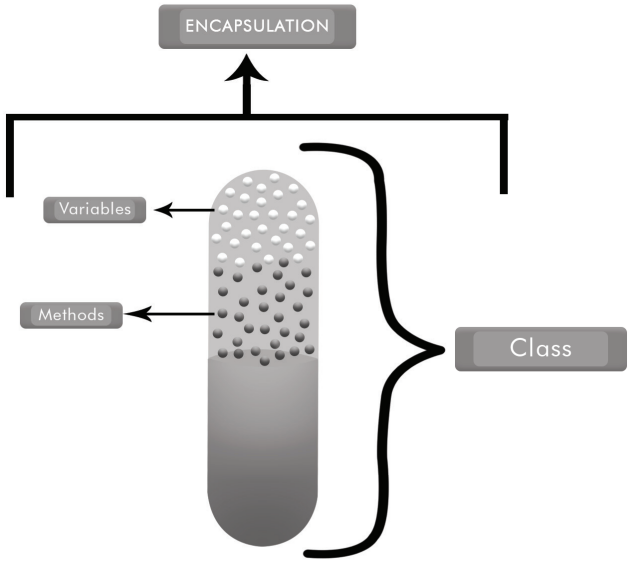

Figure 10.5 Explanation of Encapsulation.

10.3.4 Inheritance

Copying a code segment and changing its attributes creates a new record. By performing this copying and altering process multiple times, a program may have multiple instances of having a duplicate code, which requires greater maintenance and may also affect the performance of the program. This situation is resolved in OOP available in Java by implementing inheritance. This is a concept, where programmers can employ special classes which allow them to copy any attributes and the behavior of the source class. Programmers can simply override the attributes that they need to change. This creates a parent-child structure, which we will further describe when we discuss how objects relate to each other.

The main class here will be the parent one. All copied instances from this special class will be the child objects that use the same structure but have different attributes according to the requirements of the program.

Take the example of a class available for recording information of people. The different child classes may include employees, directors and managers, which may have the People class attributes as well as additional attributes that define their particular information. This principle of OOP eliminates the duplication of the code, while also removes the manual efforts required in copying and then changing all the attributes of an available class.

This creates a parent child structure as shown in Figure 10.6, which we will further describe when we discuss how objects relate to each other.

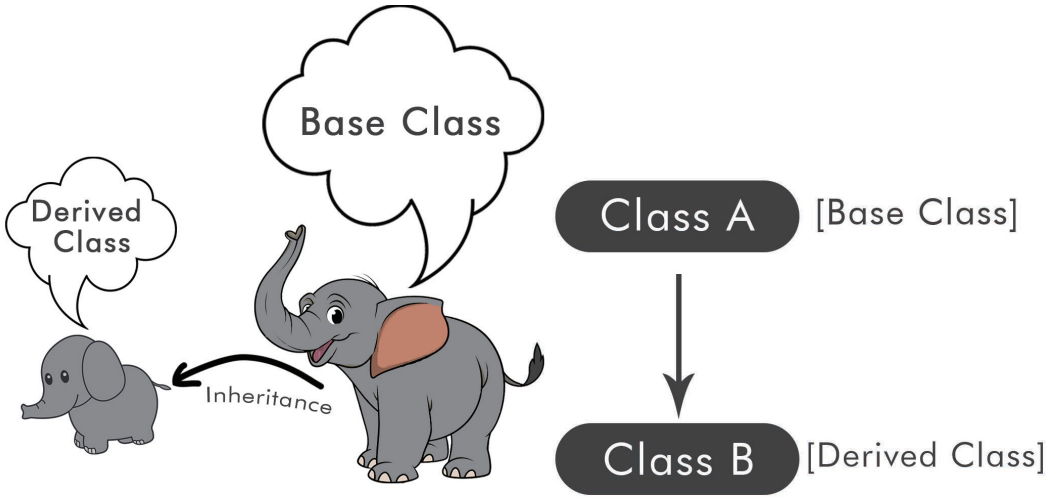

Figure 10.6 Explanation of Inheritance.

10.3.5 Polymorphism

Let us try to understand the meaning of the word polymorphism better. As per the Merriam-Webster dictionary, the word "polymorphism" can be divided into "poly" meaning "many : several : much" and "morphism" meaning "quality or state of having (such) a form". In OOP terms, it is the object's ability to take on many forms. Any class object that passes more than one IS-A test is considered polymorphic. With polymorphism, parent class object can be used to refer to the child class object. Let us define polymorphism a little differently to understand it better. Polymorphism is a special OOP concept that allows different objects of the same hierarchy to show separate behavior when they receive a similar message. The term polymorphism refers to being changed into various forms, which is what occurs when objects receive the same message.

It is a complex process often employed at higher programming levels. It allows detailed use of available data in different objects. The way objects respond is subject to how they can be employed in a specific situation.

Consider the following example shown in Figure 10.7 to understand how polymorphism is applicable for a real world situations: Walk is an action which may have different types. For example, you may walk alone, walk with someone, walk with iPod, etc. Hence, the implementation of all these will be different. This shows that one action can have many forms. This is what is known as polymorphism.

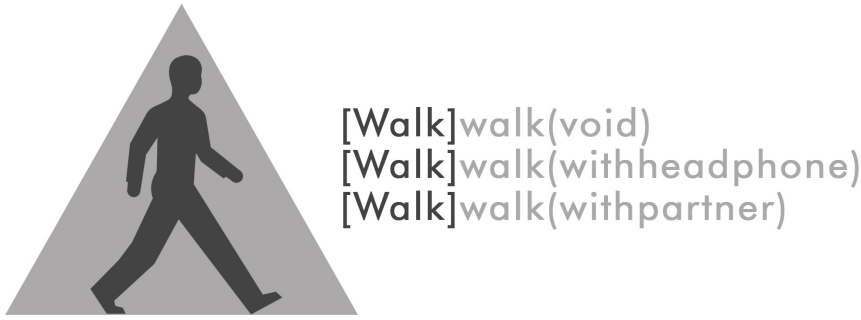

Figure 10.7 Example of Polymorphism.

Now, let us take another example shown in Figure 10.8 which is very common. In this example, you can see there is a parent class called Animal and three child classes named Dog, Cat, and Duck. They have a common method named `Sound()` that they inherit from the parent Animal class. Although they are all Animal family but each class's implementation of `Sound()` method is different like Dog says Bow Bow, Cat says Meow Meow, and Duck says Quack Quack. And you can refer any child object with Animal object.

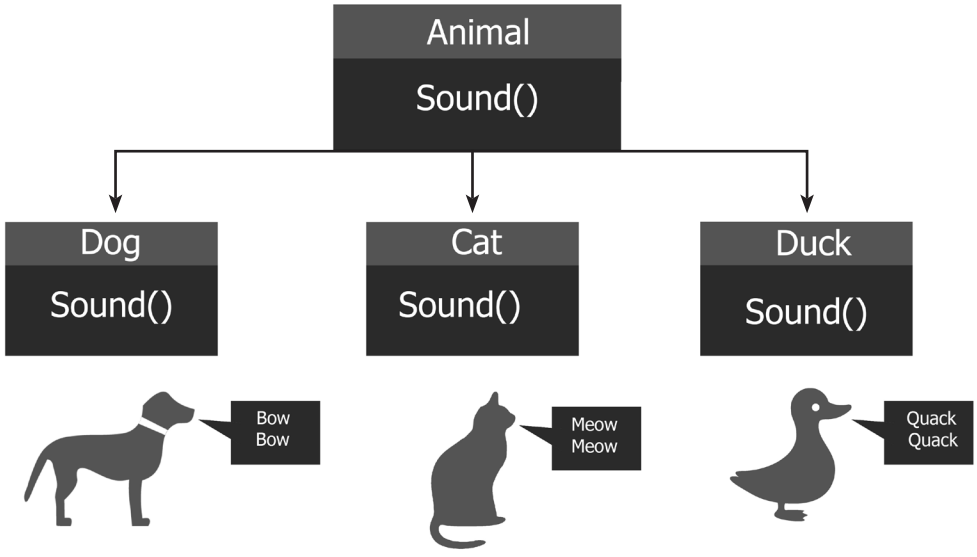

Figure 10.8 Another example of Polymorphism.

10.4 | Programming in Java

You can program in Java once you have the Java Development Kit installed on your computer. Although most text editors are capable of creating Java code files, it is important that you know where the java.exe and the javac.exe application files are present to compile the code that you can create in any editor.

You can also use an IDE for creating Java programs. We will discuss more about IDE it the last section, where we specifically describe the use of Eclipse for creating, compiling, and completing Java-based programming projects. Here, we focus on how you can start to program by first installing a JDK, learning about the language syntax, and the basic programming tools employed in Java.

10.4.1 Installing a JDK

JDK is available from Oracle for all platforms. Whether your computer runs on Windows, Linux, Mac, or Solaris, there is a JDK available free for you from the Oracle website. Once you have downloaded the JDK, you are all set to start programming. There are several programs that you can employ in addition to Java to help you program for specific environments such as preparing web pages, server controls, or database software.

The next step may be to install an IDE with which you are comfortable. We recommend using Eclipse which offers excellent programming advantages, as well as the ability to perform project management. Remember, simple Java programs and applets can be compiled and executed from the command prompt as well. However, the use of an IDE offers numerous advantages and the ease of programming that we will further discuss with Eclipse.

10.4.2 Using Libraries

The JAR files can be easily employed to enjoy the functionality offered by different Java class libraries. Your libraries are easy to add in your program. Simply add all the libraries that you intend to use on the path which describes the available Java project when using an IDE such as Eclipse. For new programmers, Java libraries contain the codes that you need to ensure that your Java program functions. They are implemented using JAR files.

A JAR library file is an archive of zipped files. Each JAR contains a collection of Java classes and other relevant sources, such as property files and icons. These files are designed to be executable when you define them in your classpath in an IDE or mention them in your programming code.

Remember, you must have the required JAR file referenced in your program if you are using different classes that are available for use in Java. Another concept is the use of executable JAR files, which allow the user to run an application without having to individually mark the Java classes which need to be initiated for a program.

In fact, the JVM must always be informed about the path of the *main()* method which is always employed in a Java program. You can easily create your own JAR files using a command:

```
jar -cvmf MANIFEST.MF myawesomeapp.jar *.class
```

This is possible by simply using a command prompt. Since we are already describing the use of Eclipse in Section 10.7, we will not focus too much attention on describing the use and the implementation of library-based classes as implementation is easy to carry out in the IDE.

10.4.3 Syntax

A key element to learn when programming with Java is the syntax of the language. The open-source programming language takes its basic syntax from C and simplifies it with the ability to objectify the code into simple blocks which are consistent and can be reused throughout the program. Remember, the language syntax is a set of rules that define how the compiler will read the information written in a text editor or an IDE.

Although Java syntax is based on C language, it does not employ global functions. It consists of data members like class level variables that always act as global variables for the class and are accessible by the class's objects in other classes.

Java employs separators of "{}" to describe a specific code block. A semicolon ";" indicates the end of a particular code element, often termed as a statement in other languages. A Java code is defined in terms of classes, which act as templates for the functionality that is required from a Java program or its different sections.

Java is a case-sensitive language, where the words, say, program and Program mean differently to the compiler and can represent separate identities. Another practice in Java syntax is that all class names must start with a capital letter. If you are using multiple words for a class, then every inner word should start with an upper-case letter. A right example of this will be to name a file as "MyFirstProgram", while a wrong practice would be to name it as "myfirstprogram".

The next syntax that you must follow is that every method, often termed as a function in other programming languages, must start with a lower-case letter. However, if it is a large name with several words, each successive word should start with an upper-case letter. An ideal example is "myMethod", while a poor one is "Mymethod" for naming methods.

Remember, your file name should be exactly the same as the class that you created. This means that according to our earlier example, you will name your file as "MyFirstProgram.java". Also remember that you need to append .java at the end of your file name to create files that show that they contain Java code and can be read by the Java compiler to create a JVM compatible Java bytecode file.

The processing always starts from the `main()` method, which is a programming syntax derived from the C language structure. A common way of using this method is in the following form:

```
public static void main(String args[])
public class HelloWorldExample {

    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```

The main program code is placed inside the braces that are present in the `main` method. All other Java components are named in a unique manner, termed as the identifiers that we will cover shortly in the subsequent programs.

Now, let us take a little detailed look at the class structure. All Java programs start by declaring a class and then defining different methods within it as required for producing the intended functionality. A common way of defining a typical class would follow this construct:

```
classAccessModifier class ClassName {
    variableAccessModifier variableDataType variableName = defaultValue;
    //Constructor declaration
    methodAccessModifier ClassName(variableDataType parameter1, variableDataType
parameter2,.....) {
        //code
    }
    methodAccessModifier methodReturnType methodName(variableDataType parameter1,
variableDataType parameter2,.....) {
//code
    }
}
```

Do not be scared of the above program as it is only to show you how Java program is structured. You do not use those keywords in writing a Java program. Let us see a quick example to understand how to use the above structure to write a program.

```
public class MyFirstClass {
    private int myFirstVariable;

    //Constructor Declaration
    public MyFirstClass() {

    }

    //Method declaration
    public int myFirstMethod(int myFirstVariable) {
        System.out.println("This is my first method which prints value of myFirstVariable :"
+ myFirstVariable);
        return myFirstVariable;
    }
}
```

This defines almost all Java programs regardless of their size, complexity, and the intended functionality. A common specification will always be to use the public access, as this allows all code in the Java program as well as an import to access the class, the variable, or the methods employed within the program.

10.4.4 Declaration Rules

You have already seen how a sample program looks like. Now is the time to see rules that are applicable to source code file creation.

1. Only one public class per source code file is permitted
2. Comments can appear anywhere, beginning or end.
3. Name of the source code file must match the name of the public class and end with "*.java*". For example, in our HelloWorldExample, we have HelloWorldExample as our public class and hence the file name must be HelloWorldExample.java.
4. Although it is not mandatory to create separate packages, it is a good practice to create separate ones. In that case, package declaration must be the first line even before import statements.
5. For any imports, import keyword must be used followed by the class full path. And these statements must appear between package declaration and class declaration.
6. There can be multiple class declarations but only one class can be classified as public. Hence, these import and package statements are applicable to all the packages and imports.
7. A file which contains the program with no public class can have any name that does not need to match any of the classes in the file. Only public class name must match with the file name and a file can have only one public class.

QUICK CHALLENGE Create a source code file, give it a name, and use the same name for the class. Mark this class as public. Now, add another class in the file and mark that as public as well. What will happen in this case?

10.4.5 Identifiers

Identifiers are the proper nouns in Java programming language, which can be used to denote any language structure. There are unique names for methods, classes, packages, interfaces, and variables in Java. In fact, every word other than the literals and language keywords can be termed as identifiers. Take our example where all names like `main`, `args`, `String`, `println`, and `MyVeryFirstJavaProgram` represent identifiers. However, they all represent different Java elements which we will describe in their relevant sections.

There are some rules to be followed when setting up unique identifiers in Java:

1. The first rule is that we cannot use a space within a name in Java. Spaces between words cannot be identified by the compiler and therefore, remain restricted.
2. Another rule is that we cannot start an identifier with a number. However, it is possible to start with an underscore, although it is not the ideal practice. There are some other characters which should not be employed such as hyphen, apostrophe, and ampersand.

There are also some set naming practices which provide a convention. Class names start with a capital letter, while methods always start with a small letter. This policy ensures that it is easy to find out a mistake in the code, while also ensuring that any programmer looking at your code can quickly identify the type of a unique element, without looking around for the neighboring code elements and statements.

Another point within this is to remember that Java identifiers are always case-sensitive, and it is not possible to keep using different cases, as the compiler will not understand a case use, and will generate an error. The last point is that Java has several keywords which it uses to represent specific meanings and literal values. You cannot use these keywords as identifiers in your Java programs.

QUICK CHALLENGE Give some examples of identifiers.

You must follow these rules when naming your identifiers:

1. All identifiers must begin with $ (currency), _ (underscore), or any alphabet (lower- or upper-case). Examples of valid identifiers are $myVariable, _myVariable, myVariable, myVariable$, my_Variable.
2. After the first character any character is allowed to name an identifier.

3. A Java keyword cannot be used as an identifier.
4. Java keywords are case-sensitive.

Java also uses modifiers. As you saw earlier, the word "public" was used to define the main method. There are two types of modifiers: (a) *access modifiers* that include public, private, protected, and default values and (b) *non-access modifiers* that include abstract, final, strictfb values.

Let us explore these modifiers next.

10.4.6 Access Modifiers

Now, let us see how access modifiers look from a microscopic point of view. When we talk about access modifiers, we talk about how much we need to restrict or allow access to a class, method, or variable.

Java offers four types of access modifiers but only three have names – public, private, and protected. The fourth one does not have any name and it is known as default or package level access. Default or Package level access means every class, method, and instance variable has access control, whether we assign or not. Now here it gets interesting: class can be of public or default access as other two modifiers do not make sense. You may declare an inner class as private.

Now, let us have a look from the class point of view. Let us see how these modifiers are useful for classes.

10.4.6.1 Class Level

1. **Default access or package level access:** As we have just seen, the default modifier does not have any modifier name before it. This modifier is known as package level modifier as the class with this type of modifier is only seen from the same package classes. For example, if the class myClass is in the package called com.fullstackdevelopment.mypackage and has no modifier in front it then this class cannot be accessed by the class yourClass which resides in other package named com.fullstackdevelopment.yourpackage. Since these two packages are different, yourClass has no access to myClass.
2. **Public access:** A class that is declared as public is accessible by all the classes from all the packages. This is the most used access modifier for a class. In the above scenario, if we change the access modifier for myClass as public, then myClass becomes available to *yourClass* even though it is residing in two different packages.

10.4.6.2 Method and Instance Variables Level

Method and instance variables can use access modifiers in the same way. As we have seen earlier, class can only use two of the access modifiers; however, methods and instance variables can use all four – Public, Protected, Default, and Private. See Figure 10.9 which shows the access modifier levels.

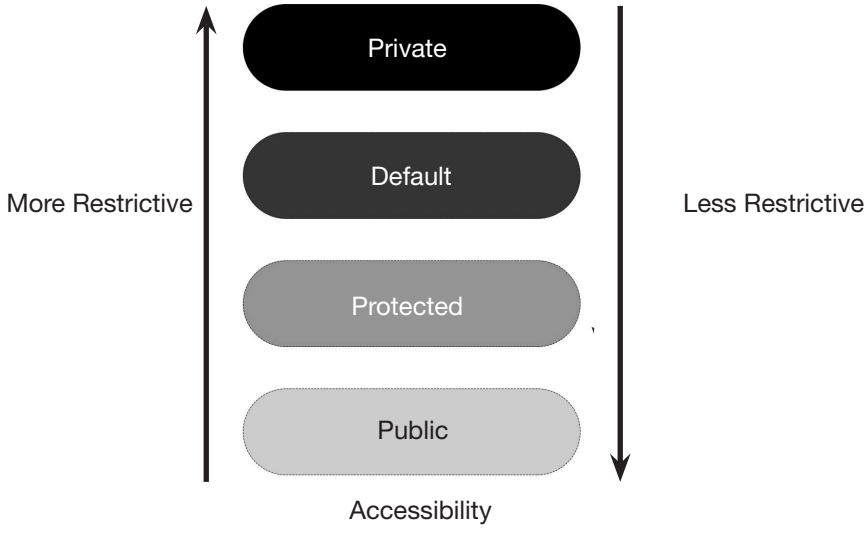

Figure 10.9 Access modifier levels.

Following modifiers are applicable to methods and variables. Public and default modifiers are similar to class modifiers that we have just seen above.

1. **Default access modifier:** When we do not declare any modifier, we get default protection which is at package level.
2. **Public access modifier:** Public access control is much wider because any class that has access to the class that contains public methods and instance variables can access these methods and instance variables.
3. **Protected access modifier:** Protected access modifier works quite similar to default modifier except for one difference that is default modifier methods and instance variables are accessible to the package only whereas protected methods and instance variables can be accessed via subclass even though subclass is in a different package.
4. **Private access modifier:** Private is the most restrictive access modifier. This modifier is only accessible by the same class members. No other class, even a subclass, can access private members of the superclass.

Table 10.2 summarizes the visibility for the access modifiers.

Table 10.2 Access Modifiers Comparison

Visibility	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, *through inheritance*	No	No
From any non-subclass class outside the package	Yes	No	No	No

To understand this better, let us see the same table in a different way:

Modifier	Class	Package	Subclass	Global
Public	✔	✔	✔	✔
Protected	✔	✔	✔	✘
Default	✔	✔	✘	✘
Private	✔	✘	✘	✘

It is important to understand the concept of modifiers and how their scope is defined. Figure 10.10 shows the modifiers scopes and how they are limiting the access for identifiers.

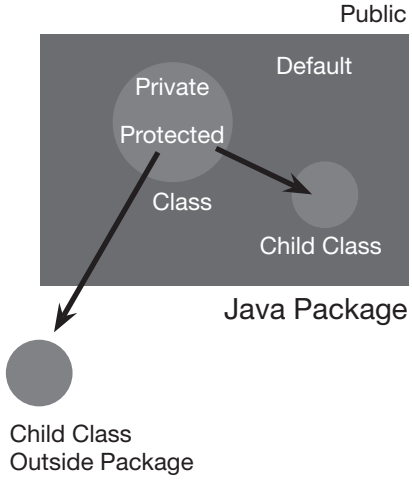

Figure 10.10 Access modifiers scopes.

10.4.7 Non-Access Modifiers

Class can also be declared with non-access modifiers such as final, abstract, or strictfp. These non-access modifiers are additional modifiers you could use with access modifiers. For example, a class can be public and final. You could also use two non-access modifiers such as final and strictfp. However, it is not at all advisable to use final and abstract. Let us see final and abstract modifiers in detail below.

1. **Final modifier:** This modifier makes the marked class non-inheritable. In other words, the class marked as final cannot be subclassed that is it cannot be extended by any other class. If tried, it will throw a compiler error. Now this might make you think, why would we even think of marking a class as final if we cannot extend it? Isn't this violating the object-oriented principal? Yes, you are right but there are circumstances when you want an absolute guarantee that the class's behavior stays intact and no one can make changes to any method. In this scenario, final comes handy. Some of the core Java libraries are final, for example, String class. Final gives the guarantee to the String class that the class functioning is intact. Example

```
public final class MyFirstClass {
}
```

   Now, no class can subclass MyFirstClass.
2. **Abstract modifier:** A class marked as an abstract can never be instantiated. The whole purpose of having an abstract class is to be subclassed. Now, why would you have a class that cannot be instantiated? One reason could be that the class is too abstract, and you want the subclasses to extend it properly. For example, a car could have some generic methods that a typical vehicle can have. But we may want very specific cars with very specific features and not generic like this abstract car. This is when abstract modifier is useful.
   If you declare a method as abstract, you must declare a class as abstract as well. However, an abstract class may have any or all non-abstract methods. Figure 10.11 shows the rules of using Abstract class.

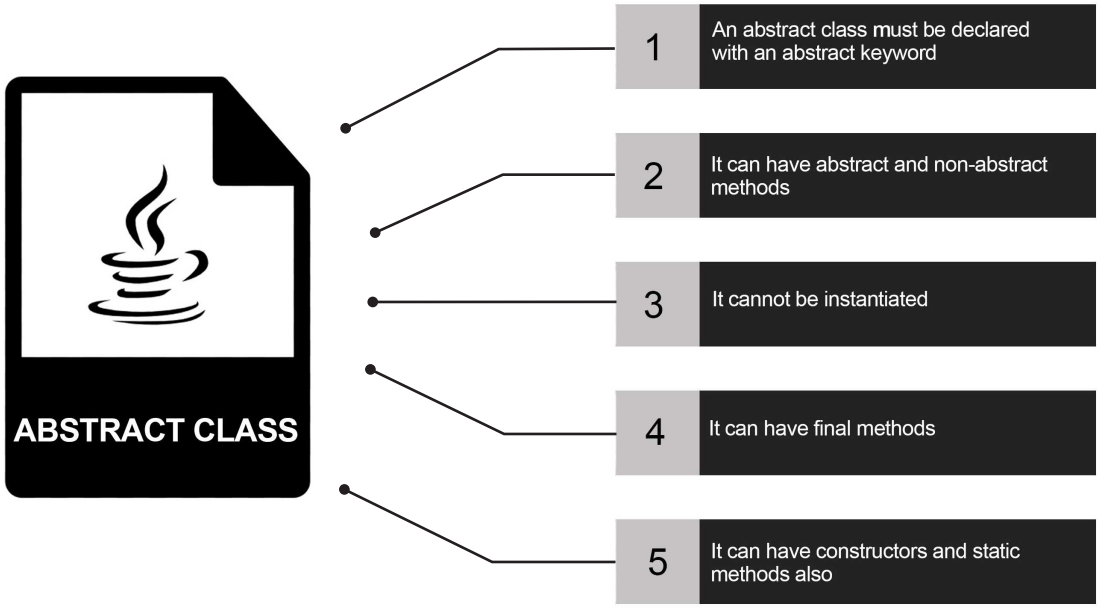

Figure 10.11 Rules for Java Abstract class.

If you declare a method as abstract, you must declare a class as abstract as well. However, an abstract class may have any or all non-abstract methods.

10.4.8 Reserved Words

As with every high-level programming language, there are several words that are reserved to represent important language commands and built-in functions. Here is a list of the words that you are not allowed to use when naming different elements in your Java program. They are reserved for Java use, and you will often employ them in programming in the same capacity:

1. abstract	18. final	35. public
2. assert	19. finally	36. return
3. Boolean	20. float	37. short
4. break	21. for	38. static
5. byte	22. goto	39. strictfp
6. case	23. if	40. super
7. catch	24. implements	41. switch
8. char	25. import	42. synchronized
9. class	26. instanceof	43. this
10. const	27. int	44. throw
11. continue	28. interface	45. throws
12. default	29. long	46. transient
13. do	30. native	47. try
14. double	31. new	48. void
15. else	32. package	49. volatile
16. enum	33. private	50. While
17. extends	34. protected	

Java also has three constructs of null, true, and false which you cannot use because they are read as literals by the language compiler. An excellent advantage of programming in an IDE is that it uses a color scheme that quickly allows you to identify that you have written a Java reserved word in your code. See the following example which shows the reserved words in purple color that includes package, class, private, int, public, and return.

```
1  package com.fullstackdevelopmentwithmayurramgir;
2
3  public class MyFirstClass {
4      private int myFirstVariable;
5
6      //Constructor Declaration
7      public MyFirstClass() {
8
9      }
10
11     //Method declaration
12     public int myFirstMethod(int myFirstVariable) {
13         System.out.println("This is my first method which prints value of myFirstVariable :" + myFirstVariable);
14         return myFirstVariable;
15     }
16
17 }
```

10.4.9 The Keyword "this"

The keyword "this" refers to the current object. Following list shows the use of **this** in any Java program.

1. Current class instance variable can be referred by **this**.
2. Current class method can be invoked implicity by **this**.
3. Current class constructor can be invoked by **this**.

4. A method argument list can contain **this**.
5. A constructor argument list can contain **this**.
6. A method can return **this** which refers to the current class instance.

10.4.10 Classes and Objects

Java classes and objects form the basis of the OOP concept in the language. The first principle that defines a Java class is that it should have singular functionality. This allows the use of class or particular objects whenever a particular need arises without having to write new code. This principal is called as "Cohesion". Cohesion is the degree to which a one particular class has a single, exclusive purpose. If a class has high cohesion, it means the class is focused on a single functionality and only useful to serve a particular purpose.

To understand this better, we give an example of a Student class and its objects. Student is a class which only focuses on student-related functions and it has student-related states and behaviors. All the objects that are deriving from this class get these student-related states and behaviors and do not carry any other functionality.

Figure 10.12 shows a Student class that has states like Name, Age, Color, and Sex. Student class also has behaviors like Eating, Drinking, and Running. Every object derived from this particular Student class gets these states and behaviors. The left side of Figure 10.12 shows Objects which have their states defined like first object has Name as Roshni, Age as 11, Color as Dark and Sex as Female. Similarly, other objects get the states and can have their own values for those states.

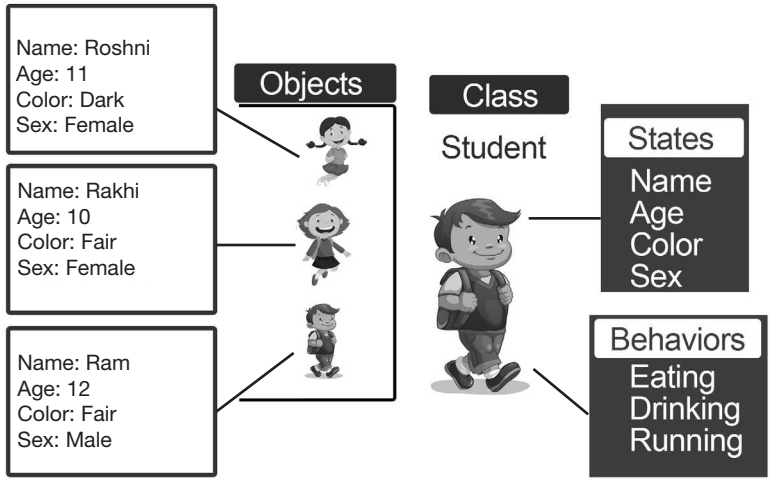

Figure 10.12 Objects and Class example with States and Behaviors.

Another property that a Java class has is an open, closed design. This is a design which describes that the classes and the methods present in them must remain closed to prevent modification. However, they should be open to an extension where the current code present in the class can be used to implement further functionality in a Java program.

A Java class works as a blueprint for another important concept, which is the use of objects that function as discrete blocks of code in the programming. Your Java application uses object instances. It is designed to carry out the functionality required of it, in different sections of the program. Java packages control the way naming conflicts are resolved, and therefore a namespace is always uniquely created when compiling a program in Java.

Java packages are named using a standardized scheme. This improves control over generated classes and allows programmers to use the available packages and libraries. In fact, if you name all your classes in a unique manner, you will never have to mention a package to describe them.

The working part in Java programming is an object. It is often a particular class case which has its own definition and behavior according to the inputs it receives. The Java objects often follow a simple hierarchal structure, where a parent object serves as the basis for other objects (child) that come under it. This structural exercise allows you to create a set of objects that have the same definition but can show different behaviors. A key feature of Java objects is their ability to communicate with each other. This ensures that a program runs in a coordinated manner and can perform tasks in a systematic manner, especially in relation to the specific case that may appear during each program execution.

Java objects must perform a definite set of activities. They should only declare or hold their own data in an ideal scenario. However, they may need access to other objects if they are employed in carrying out the defined activities within their definition. An object must only have a minimalistic set of dependencies that are sufficient for it to perform its activities.

Take the example of an object that defines a person. Such an object in line with the Java object definition will have a set of two elements – the attributes and the object behavior. The attributes are often termed as the nouns that define an object. A Person object may have attributes such as age, name, contact details, gender, and other elements that may present the definition of the described object.

The behavior of an object may be best understood as the action verbs that it can perform. Let us follow our example of a Person object. A behavior would be to use the available attributes to calculate the BMI. Another verb in this manner would be to print all attributes for a particular object instance. The state of the current attributes can also be found as a behavior, where the object can return the current values using a String.

10.5 | Java Packages

Most Java programs start by mentioning a Java package. Java packages represent collection of classes and their related code elements. Packages also provide the option of using access modifiers. A package is present at the start of the program file and provides a dedicated path for the storage of the program class.

10.5.1 Structure

Defining a class is important. You use it to define various objects, and in turn, Java application will run object instances on execution. There is an import statement present in a class definition as well. This allows the compiler to find the classes that you use in your code. One class may possibly use other classes, and it necessitates that the compiler is aware of where to read them for execution. The structure of the import is as follows:

```
import ClassNeedsImport;
```

The class name should always include its package as well because it is required for an independent qualification. There are times where you may need multiple classes that you are using in your non-trivial class. It would not be an ideal practice to name every individual class separately. You can import a set of classes as well. By using a ".*" after a package name in the import statement, you can import every available class from the package for use in the program. Here is an example:

```
import com.javapackage.*;
```

A good practice is to only import the classes that you need by using their qualified names. This keeps your code easy to read and ensures that you are following the minimalistic structure, which is the basis of using an OOP language like Java. For any object that you need to create, you must first declare its class. This works out like creating a template which can then be employed multiple times to provide the same functionality.

A common class declaration is defined in the following manner:

```
classAccessModifier class ClassName {
    variableAccessModifier variableDataType variableName = defaultValue;
    //Constructor declaration
    methodAccessModifier ClassName(variableDataType parameter1, variableDataType
parameter2,.....) {
        //code
    }
    methodAccessModifier methodReturnType methodName(variableDataType parameter1,
variableDataType parameter2,.....) {
        //code
    }
}
```

The specifier remains public. The class starts with an upper-case letter. This allows all programmers who may look at your code to easily understand the required functionality of the class, as well as the objects that are created through it. A class heavily depends on the variables that allow each instance of the class to be distinguishable and fully separate from one another.

Figure 10.13 shows how classes are defined. As you can see in the left box, a class definition contains variable, methods, and statements inside methods. And in the right box, you can see the code syntax for defining a class. In this example, 'type' can be any primitive, non-primitive or object type.

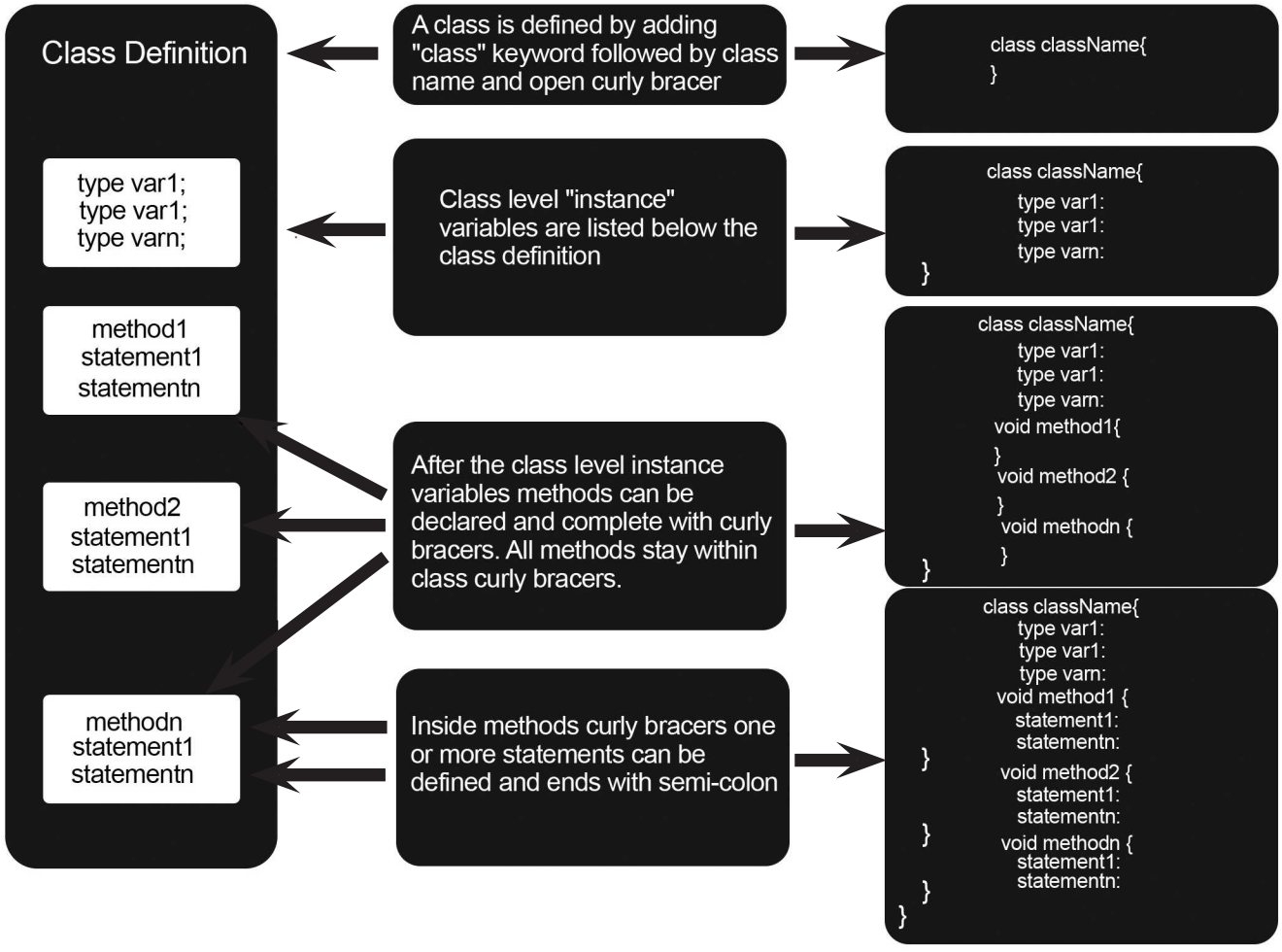

Figure 10.13 Class structure and elements.

10.5.2 Variables

Each class is defined by a set of class variables. They declare the specific state of the class in its different instances. This enables the use of the class elements in a variety of settings. The first element in a variable is an access modifier. This can have any state – from the group of public, private, protected, and no specifier. If no specifier is used, the variable is available for the entire Java package.

The next part is the data type, which defines whether it is a primitive class variable or belongs to another type. The variable name is the next part, which should follow a camel case. This is a convention which starts with a lower-case letter. Programmers can also define an initial value for the variable. This is just like the initialization that you can use with variables that are created in other programming languages. The compiler sets up an initiating value for the class variables if they are not predefined. A class would often have a set of variables in the following manner:

```
public class GymCustomer {
    //In the following line 'private' is an access modifier, 'int' is a primitive data
type and 'age' is a variable
    private int age;
    private int height;
    private String name;
    private String gender;
}
```

The above code contains a class with the name GymCustomer that has a set of attributes in the form of variables. There are two data types mentioned in these variables. The first are the integers which describe the use of whole numbers, while the second one is String designed to record character strings such as the name, gender, and skin tone of the GymCustomer class.

10.5.3 Methods

The information about a particular class is incomplete without the use of methods. Methods are behaviors that the class objects can show. There are two categories of methods. The first category is that of the constructors. All other methods such as void and the ones which return value form the second category and are used to signify any customized behavior which is required of a particular class instance.

As visible with our earlier example, methods also need to have their access specified. They also require a type of return, which signifies the value that they will produce. The name is mentioned next, while the parentheses contain the arguments required for the intended functionality. The complete text that contains a collection of these details is termed as the signature of the method. The infographic shown in Figure 10.14 can help us understand this concept a little better.

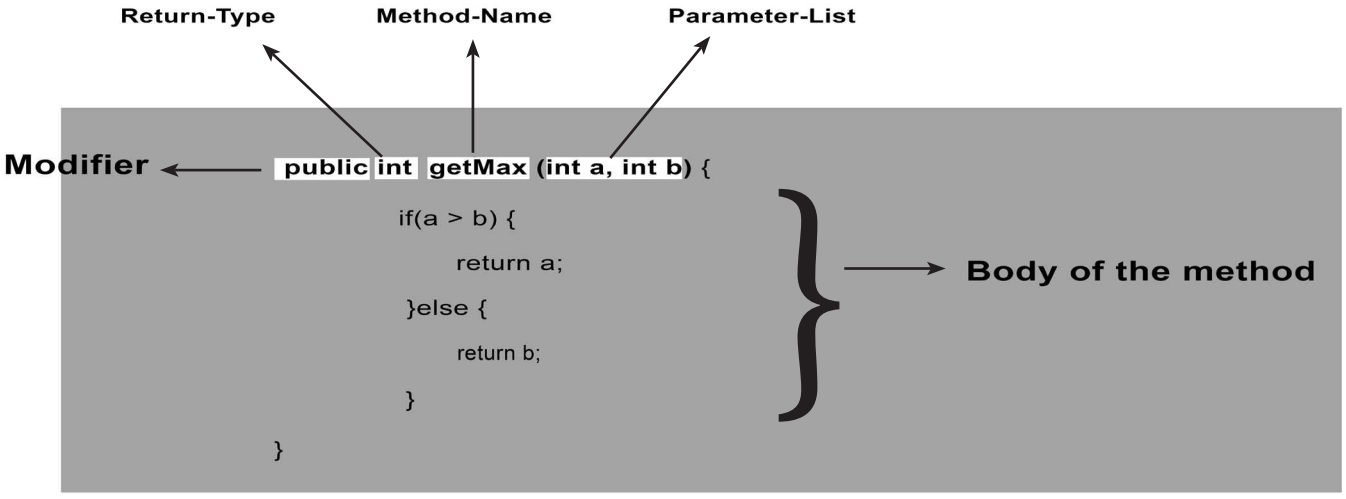

Figure 10.14 Methods.

10.5.4 Constructors

Constructors are methods that describe the class in the form of different statements. The compiler automatically uses a no argument constructor for your instantiating needs. This makes them optional. However, if you use a customized constructor, then the compiler will not create one for the specific purpose.

A constructor is always named after its class. It looks like a method but without a return type. Default constructor is the one which does not take any parameters and non-default constructor is the one which takes parameters in other words argument list. Constructor can have similar type of access modifiers as methods can have. If no modifier is given then it is given a package level access. If the constructor in a class is defined as private, no class can instantiate this class as it only gives class level access. The only way to create objects of this class is by having a factory method in the class that calls the private constructor. You can also use this constructor to create the initial attributes. Let us move forward with our earlier example:

```
package java11.fundamentals.chapter10;
public class GymCustomer {
    private int age;
    private int height;
    private String name;
    private String gender;
    public GymCustomer() {
        // Nothing happens here
    }
}
```

In the above example, we have seen "no argument" constructor which is also called as default constructor. The following example shows a different type of constructor where we accept the default values for the member variables while object creation.

```
package java11.fundamentals.chapter10;
public class GymCustomer {
    private int age;
    private int height;
    private String name;
    private String gender;
    public GymCustomer() {
        // Nothing happens here
    }
    public GymCustomer (int age, int height, String name, String gender) {
        this.age = age;
        this.height = height;
        this.name = name;
        this.gender = gender;
    }
    public void printGymCustomerData(){
        System.out.println("Name : " + this.name + " - Gender : " + this.gender + " - Age
: " + this.age + " - Height : " + this.height);
    }
}
```

Can you define constructor as private?

This constructor uses "this" keyword which signifies this object. It allows the use of variables that have the same name. The age, for example, is a class variable, but it is also used as a parameter in the class constructor. The use of "this" allows the compiler to differentiate between these elements with the same name. Following is the example of using these constructors.

```
package java11.fundamentals.chapter10;
public class GymCustomerTest {
    public static void main(String args[]){
        //this will call the no argument constructor to create GymCustomer object
        GymCustomer gc = new GymCustomer();
        // this will call the argument constructor with default values set for the member
variables
        GymCustomer gcc = new GymCustomer(45,5,"Mark Smith","Male");
        gcc.printGymCustomerData();
    }
}
```

The above program will print the Gym customer data like name, height, age, and gender. See the following output.

```
Name : Mark Smith - Gender : Male - Age : 45 - Height : 5
```

10.5.4.1 "this" keyword

The "this" keyword is useful to call other constructors in the same class. This type of invocation calls as explicit constructor invocation. See the following example which shows three constructors and first two constructors call the third one explicitly.

```
package java11.fundamentals.chapter10;
public class ThisExample {
    private int x, y;
    private int a, b;
    public ThisExample() {
        this(0, 0, 1, 1);
    }
    public ThisExample(int a, int b) {
        this(0, 0, a, b);
    }
    public ThisExample(int x, int y, int a, int b) {
        this.x = x;
        this.y = y;
        this.a = a;
        this.b = b;
        System.out.println("Printing from the Third Constructor - x : " + x + " - y : " +
y + " - a : " + a + " - b : " + b);
    }
    public static void main(String args[]) {
        ThisExample te = new ThisExample();
        ThisExample te2 = new ThisExample(2,2);
        ThisExample te3 = new ThisExample(3,3,3,3);
    }
}
```

The above code produces the following result. This shows how the call passes through the constructor using "this" keyword.

```
Printing from the Third Constructor - x : 0 - y : 0 - a : 1 - b : 1
Printing from the Third Constructor - x : 0 - y : 0 - a : 2 - b : 2
Printing from the Third Constructor - x : 3 - y : 3 - a : 3 - b : 3
```

Flash Quiz

What will happen if you do not define any constructor?

10.5.4.2 "super" keyword

"super" keyword refers to the super class. "super" is used to invoke the superclass's constructor. It is only allowed to use on the first line or else compiler will complain. See the following example which shows the use of "super" keyword.

```
package java11.fundamentals.chapter10;
public class SuperExample {
    SuperExample(){
        System.out.println("This is from the SuperExample Constructor");
    }
    public static void main(String args[]) {
        System.out.println("Inside the main method");
        Child c = new Child();

    }
}
class Child extends SuperExample {
Child() {
super();
System.out.println("Child Constructor");
    }
}
```

The above program produces the following output;

```
Inside the main method
This is from the SuperExample Constructor
Child Constructor
```

In this example, you can see that from the Child constructor we have made a call to `super()` which calls the super class constructor from SuperExample.

Table 10.3 shows how constructors are different from methods in various areas.

Table 10.3 Difference between constructors and methods

	Constructors	Methods
Use	For creating instances of a class	Group code to
Applicable Modifiers	Constructors can be Private, Public, Default, and Protected but cannot use abstract, final, native, static, or synchronized	Methods can use private, public, default, protected abstract, final, native, static, or synchronized
Name	It must be exactly same as the class name.	Any name other than the class name can be used.
This	this refers to another constructor in the same class. It must be used on the first line in the constructor declaration.	this refers to an instance of the current class. this cannot be used in static method.
super	super refers to the parent class constructor. It must be used on the first line in the constructor declaration.	super is useful to call an overridden method in the parent class.
Inheritance	It is not applicable to constructors as they cannot be inherited.	Methods can be inherited
Return type	None	Native data types like int, float, double etc., Native objects like String, Map, List etc., or any other built-in and user defined objects.

10.5.5 Other Methods

The GymCustomer object is ideal to understand the basic syntax of the language. However, you need more behavior implementation. This requires use of more methods that truly create the functionality that a class object needs. Other methods are different because they can have any name (other than the class).

Some important notations, though optional, are great for reviewing your code. You should always start the name with a lower-case letter and avoid using numbers as much as possible. Other methods also allow you to mention a return type, which produces specific functionality for the object. If we continue with our earlier example, we can show how it is possible to use methods other than the constructors. Here, we have omitted the class details and are showing the use of a method after the initial class variables:

```
public String getGender() { return gender; }
public void setGender (String value) { gender = value; }
    // May use any combinations like this for information….
}
```

This is the use of an instance where you use two particular methods. The first is the getter method that fetches the value of a variable (attribute), while the setter then allows you to modify that value. You can do this for any attribute which is defined in the class definition. Remember, it is important to mention the type of return in this use. Since the getter method brings a string, its datatype is defined for it. On the other hand, the setter does not return anything, which is then clearly stated using the reserved word "void".

You can generally use static and instance methods. The static methods are often termed as class methods because they change the behavior at a class level, which is independent of the individual class objects. The instance methods, as their name suggests, are designed to work according to a particular object instance.

Static methods are excellent because they provide utility and ability to offer global functionality, just like the C language. The code used in the static method remains confined within the class that defines this static method. Take the example of a Logger class which is often used to output the information. The static method of `getLogger()` is employed whenever you need the functionality of the particular class.

A static method uses a different syntax since it directly applies to a class rather than individual objects. It is possible to use the name of the class within its use as follows:

```
Logger l = Logger.getLogger("NewLogging");
```

The first Logger is the class name and the `getLogger("NewLogging")` is the static method used. This means that there is no need to create or mention an object instance. The class name is sufficient for creating this type of method use.

Which return type can you use to not return anything from a method?

10.5.6 Programming Examples

In this section, we will discuss some simple examples that will show the basic Java elements that we have described in this particular section. These are simple Java programs that will help you understand the basic functionality that this object-oriented language offers to programmers. Let us start with the typical Hello World program:

```
public class HelloWorld
{
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

This is a simple program with the Java class of Hello World. It uses a single method of *main()* which describes the use of a string output. There is only one functionality present in the method, which is to print the message of "Hello World".

```
Hello World
```

Now from this simple example, let us move to something more complex, which uses objects:

```
public class LearnJavaProgramming {
    public void printFeedback(){
        System.out.println("I love Java");
    }
}
public class MyFirstProgram {
    public static void main(String[] args) {
        // creates ljp1 object of LearnJavaProgramming class
        LearnJavaProgramming ljp1 = new LearnJavaProgramming();
        // creates ljp2 object of LearnJavaProgramming class
        LearnJavaProgramming ljp2 = new LearnJavaProgramming();
    }
}
```

This is a simple example where a class MyFirstProgram is initiated and then two objects of the same class are added to the `main()` method.

The next example describes the use of methods with objects which bring the real use of the Java structure and its programming methods. The program below from the same source of the above program is an extension that shows the use of methods and calling them to control the class objects in Java:

```
public class LearnJavaProgramming {
    private String feedback;
    public void printFeedback(){
        System.out.println(feedback);
    }
    public void setFeedback(String feedback){
        this.feedback = feedback;
    }
}
public class MyFirstProgram {
    public static void main(String[] args) {
        // creates ljp1 object of LearnJavaProgramming class
        LearnJavaProgramming ljp1 = new LearnJavaProgramming();
        // creates ljp2 object of LearnJavaProgramming class
        LearnJavaProgramming ljp2 = new LearnJavaProgramming();
        ljp1.setFeedback("First object feedback - I love Java");
        ljp2.setFeedback("Second object feedback - I love Java");
        ljp1.printFeedback();
        ljp2.printFeedback();
    }
}
```

Now you can print the feedback of the two objects. Since the first object ljp1 is setting its feedback to "First object feedback – I love Java", it will print the same when `ljp1.printFeedback()` gets called. The same is true for the second object where `ljp2.printFeedback()` will print "Second object feedback – I love Java".

```
First object feedback - I love Java
Second object feedback - I love Java
```

10.6 | New Features in Java 9

There are several new features in Java 9. You can start with the basic that we have covered till now and learn about the advantages of the new version with the information available in this section. Even if you are a beginner, learning about these new features will allow you to focus on the excellent elements in the new version, and include them in your normal programming routines. In the following subsections we will give brief details of the new additions in the latest Java version.

10.6.1 Java Shell

Java was traditionally a language that required programs to be compiled before execution. With the arrival of the Read–Eval–Print–Loop (RELP) functionality through the JShell tool in Java 9, it is now possible to run simple Java code at a jshell console, and instantly execute the output.

Please see the following screenshot of a JShell console, which shows how easy it is to create variables, assign values to the variables, create methods, use the assigned variables in the newly created methods, etc.

```
Mayurs-MacBook-Pro:~   Mayur$ jshell
|  Welcome to JShell -- Version 11.0.1
|  For an introduction type: /help intro

jshell> String myName = "Mayur Ramgir"
myName ==> "Mayur Ramgir"

jshell> String HelloMethod(String s) {
   ...> return "Hello " + s;
   ...> }
|  created method HelloMethod(String)

jshell> HelloMethod(myName)
$3 ==> "Hello Mayur Ramgir"

jshell>
```

In the above program, you can see, we started jshell by typing jshell command on the command prompt. With this we get into the jshell prompt. On that, we have declared variable as “String myName” and assign value as “Mayur Ramgir”. It’s the same way we type inside a method as local variable or in a class as class level variable. Upon pressing enter key on the keyboard, jshell registers the variable which we can use later. After that we have created a method “String HelloMethod(String s)” and added a return string “return ‘Hello’ + s”. Again, upon pressing enter key on your keyboard, jshell registers the method. In the next line, we have used that method and passed the variable “myName” that we have created earlier. Upon pressing enter, it shows the result as “Hello Mayur Ramgir”.

The new Java Shell is great when you are learning about the different aspects of the programming language. You can directly learn about the behavior of an API by simply running different code elements through the JShell. It is present as an API to offer you excellent REPL functionality and bridge the gap that the previous versions had.

10.6.2 Multi-Release JAR Files

Another new feature is that now the JAR format allows use of multiple versions of the same class files to be placed in a single JAR archive file. These class files have different versions for different releases. This is termed as the option of supporting multi-release JAR files within the archive. It allows the use of a single library file with multiple versions and different programming scenarios in the form of Java platform releases.

10.6.3 Platform Support for Legacy Versions

Java 9 has an enhanced javac compiler with the ability to run programs that may have been prepared using the earlier versions of the programming platform. You can employ the -target or the -source options in Java 9, which can cause problems when

compiling for different platforms. However, Java 9 allows the use of release option that ensures that you cannot use APIs that are not supported by the current compiler.

10.6.4 Java Linker

The JEP 220 defines the jlink tool which got introduced in Java 9 that empowers the assembly and the linking process of classes and libraries. In many situations, we don't need an entire JRE to run our programs. For example, to run a simple calculator application which can only performs addition, subtraction, division and multiplication operations. For such applications, we don't need all the classes that come with pre-built JRE which are nearly 4300 plus in numbers. In this case, we can only get the classes which are absolutely needed that can save on memory consumed by runtime. This can be done by JLink. It optimizes modules and ensures that their dependencies can produce a reliable runtime image for functionality. Java Linker is a tool that provides transformation during the assembly and allows the use of different image formats.

10.6.5 Hash Algorithms

Java 9 now supports the powerful SHA-3 hash algorithms that are defined in the NIST FIPS 202. It also supports cryptographic elements that significantly improve the security that Java 9 offers over the previous versions. The java.security.MessageDigest API describes the use of the following hash algorithms as supported by the new platform:

1. SHA3-224
2. SHA3-256
3. SHA3-384
4. SHA3-512

10.6.6 Modular Java Packaging

Java 9 is entirely based on the use of modular packages. This allows for creating a robust programming environment where you can easily create the required programming solution. You can implement a new phase of linking that can separate the execution runtime image phase from the compilation time. This is possible with the jlink tool that we have already described in Section 10.6.4.

There are also options that can be added to specify the position of different modules when using the linker, the compiler, or the JVM. The modular system now also provides access to module-based JAR files. These are the files which you can add in a module-info.class form and put them in the root directory of the platform.

There is now the JMOD format available which is a new package format that provides significant improvements over the currently available JAR format. It can include configuration files as well as native code elements and is activated using the jmod tool.

The JDK is presented in the form of several modules. This allows you to use a variety of configurations, according to your runtime environment needs. The modules can offer compact profile performance as described in the previous Java version. There is also a new scheme which at runtime allows to add new Java elements to the image without affecting its format.

Some redundant tools like java.*, sun.misc, sun.reflect, java.awt.peer, etc. are removed to simplify various JDK APIs and it is now possible to employ various modular options directly from the command prompt by accessing the directory of the installed JDK.

There are five types of modules in the new module system:

1. **System Modules:** These are the Java SE and JDK modules which can be viewed by list-modules command shown below.

```
java --list-modules
```

Run the above command in a terminal window. Upon running the command we get the following output.

```
Mayurs-MacBook-Pro:~  Mayur$ java --list-modules
java.base@11.0.1
java.compiler@11.0.1
java.datatransfer@11.0.1
java.desktop@11.0.1
java.instrument@11.0.1
java.logging@11.0.1
java.management@11.0.1
java.management.rmi@11.0.1
java.naming@11.0.1
java.net.http@11.0.1
java.prefs@11.0.1
java.rmi@11.0.1
java.scripting@11.0.1
java.se@11.0.1
java.security.jgss@11.0.1
java.security.sasl@11.0.1
java.smartcardio@11.0.1
java.sql@11.0.1
java.sql.rowset@11.0.1
java.transaction.xa@11.0.1
java.xml@11.0.1
java.xml.crypto@11.0.1
jdk.accessibility@11.0.1
jdk.aot@11.0.1
jdk.attach@11.0.1
jdk.charsets@11.0.1
jdk.compiler@11.0.1
jdk.crypto.cryptoki@11.0.1
jdk.crypto.ec@11.0.1
jdk.dynalink@11.0.1
jdk.editpad@11.0.1
jdk.hotspot.agent@11.0.1
jdk.httpserver@11.0.1
jdk.internal.ed@11.0.1
jdk.internal.jvmstat@11.0.1
jdk.internal.le@11.0.1
jdk.internal.opt@11.0.1
jdk.internal.vm.ci@11.0.1
jdk.internal.vm.compiler@11.0.1
jdk.internal.vm.compiler.management@11.0.1
jdk.jartool@11.0.1
jdk.javadoc@11.0.1
jdk.jcmd@11.0.1
jdk.jconsole@11.0.1
jdk.jdeps@11.0.1
jdk.jdi@11.0.1
jdk.jdwp.agent@11.0.1
jdk.jfr@11.0.1
jdk.jlink@11.0.1
jdk.jshell@11.0.1
jdk.jsobject@11.0.1
jdk.jstatd@11.0.1
jdk.localedata@11.0.1
jdk.management@11.0.1
jdk.management.agent@11.0.1
jdk.management.jfr@11.0.1
jdk.naming.dns@11.0.1
jdk.naming.rmi@11.0.1
jdk.net@11.0.1
jdk.pack@11.0.1
jdk.rmic@11.0.1
jdk.scripting.nashorn@11.0.1
jdk.scripting.nashorn.shell@11.0.1
jdk.sctp@11.0.1
jdk.security.auth@11.0.1
jdk.security.jgss@11.0.1
jdk.unsupported@11.0.1
jdk.unsupported.desktop@11.0.1
jdk.xml.dom@11.0.1
jdk.zipfs@11.0.1
```

The above list of modules are grouped into four major groups like java, javafx, jdk and Oracle.

2. **Java modules:** These modules contain the implementation classes of the core SE language specification.

   JavaFX Modules: These are the FX UI libraries
   JDK Modules: These are the modules needed by JDK
   Oracle Modules: These are the oracle specific modules

3. **Application modules:** These are the modules chosen by us that are needed for the execution of our program. This required list defined in the module-info.class file.
4. **Automatic modules:** These are the modules which come from the users and not included in the official build. The names of these modules are derived from the name of the jar file. Automatic modules get read only access to the other modules.
5. **Unnamed module:** These are the modules like classes or JARs which are loaded into the classpath directly but not in the module path.

10.6.6.1 Problems Solved by Modular Java Packaging

There are various benefits of Modular Java Packaging as it tends to solve a lot of problems programmers were facing earlier. Some of these problems are listed below.

1. **Memory intensive JDK:** JDK comes with a lot of modules which may not be needed for your application need. A lot of memory is wasted on these modules as loading them adds no benefit to your application at all. Here, Modular Java Packaging helps to group the only needed modules.
2. **Unwanted exposure of modules:** Many times we want to hide or prevent the exposure of some of the modules to the outside modules. We may want to restrict the use of these modules to the internal modules from the same module package. These modules stay hidden or encapsulated. This is what you can achieve with Modular Java Packaging as only the required modules can be exported which will be visible to every other module.
3. **Runtime failure due to missing module or jar or class:** Prior to Java 9, the missing classes were not detected until the application is actually trying to use them. This was leading to the runtime error. With Java 9 Modular Java Packaging, even Java applications must be packaged as Java modules too. This makes the application to specify all the modules it expected. This enables the Java VM to check the entire module dependency graph for the application. If VM does not find all the dependent modules at start up then it reports the missing modules and shuts down.

10.6.7 Tuning Improvements

Java 9 offers improved tuning improvements. Redundant garbage collection combinations are removed to simplify the use of customized heap recollection. The G1GC that we define in the next subsection is placed as the main GC method, with several tuning parameters placed to improve its performance as the automatic garbage collection choice.

Concurrent Mark Sweep (CMS) now cannot run in the foreground as it is simply not required. The flags that govern the foreground CMS status are now also removed, improving the overall process of performing garbage collection through a reduced, but an empowered set of GC tools. All tools that lost their functionality in the previous version are now completely removed, which increases the efficiency of the current GC method and the available tuning options.

However, the current tuning improvements require programmers to learn more about G1GC, as they will now be using it if they do not want to perform manual tuning functions on a wide scale. The G1GC improves tuning because now it can make better decisions whether to improve memory usage or the processing ability during each cycle of execution.

10.6.8 G1GC as Default Collector

Garbage First Garbage Collector (G1GC) is now the primary GC method, which is automatically activated when programming in Java 9. It is a collector that is designed to provide low-pause collections. This is essential because the

modern Java programming focus is on preparing cloud-enabled applications. These applications must use huge memory heaps while allowing users to suffer from as little downtime as possible when performing an image execution. The G1GC will offer a better stop response for most Java 9 program users, with its ability to alter processing load and memory availability to reduce the collection pauses as much as possible, without causing a problem in the reliability of the execution of the program.

The G1GC is now set up to provide advanced tuning facility. It is possible to change its parameters to ensure that you can use it in the best manner, in the context of your specific program created using Java 9.

10.6.9 Process API Updates

The API process updates provide a lot more power to the Java 9 platform. The ProcessHandle is a new class that now allows programmers to learn about the parameters of the native processes that occur in the programming language. You can get information about the CPU processing time, the parent process, and the descendants that appear from the main process chain.

There is the .onExit method available in this class that allows setting of different mechanisms that can occur when this particular process exits during the runtime execution.

The new API updates allow the developers to obtain the native process ID and then use it to perform functions without turning to the use of native code that may limit the performance of a particular program. The Process class is now full of new methods that are capable of offering the improved insight into a Java stream such as interface:dropWhile, takeWhile, ofNullable, etc.

It is now possible to run the optional functions that offer you more functionality. You can use the `getPiD()` method to get the ID of the process which is currently in use. The method now returns the data type (literal) of long type, to allow for larger ID numbers. These new additions empower developers to use better programming aids and ensure that they can employ the information of the available Java processes to create dynamic programs that produce effective results in a real-time environment.

10.6.10 XML Catalog

Java 9 provides a new, standardized XML Catalog in the form of an API. This Java 9 API supports the powerful XML Catalogs that are described by version 1.1 of the Organization for the Advancement of Structured Information Standards (OASIS). This API allows programmers to get the definitions of XML catalogs and can deliver improved performance when using a Java program with the JAXP processors that run the resolver abstraction tools.

The applications that currently use the previous libraries or specific applications to provide the XML support functionality now have to turn their attention towards the new API. This will allow these applications to get more power and gain the ability to employ the new methods and features that are available in the highly improved XML Catalogs.

The XML was an excellent choice for programmers who needed to place data elements on web-based applications. The modern API allows the creation of catalogs to support local Java programs. It is easy to use and ensures that XML processors present in the JDK can use the best features of the available environment to optimize the processing XML based data elements.

The main purpose of this API remains to ensure that the use of external resources is easily implemented within Java programs and does not disturb the program behavior. It improves program performance, enhances the availability of the resources, and significantly improves the security of the platform when used with trusted XML sources.

10.7 | Eclipse IDE for Programming

Eclipse is an excellent IDE that is most widely used by the Java Community. It provides an ideal workplace where you can create your Java projects and work on them in a systematic manner. There is a plug-in system available as well. This allows the addition of external elements that can significantly improve the functionality of the platform.

Eclipse is designed for creating Java applications, but installing different plug-ins allows you to implement common programming languages such as JavaScript, Fortran, C, C++, Ada, PHP, Python, Ruby, and several others. The creation of

packages and documents is possible for the software suite of Mathematica. Common development tools are usually set up for Java and Scala.

You can install Eclipse IDE that provides powerful tools for computer programming and the ability to employ an XML editor and implement the functionality offered by the Maven project management tool. A typical package comes with all the essential tools that you need to start working on your complex Java projects. Here, we will present some basic functions and elements of this programming environment that will allow you to start your journey towards becoming a successful Java 9 programmer.

10.7.1 Advantages of Using Eclipse

As any IDE, eclipse provides a favorable environment for your chosen programming language. Eclipse offers support for languages other than Java as well. However, it is widely used for Java and J2EE related development. Following are some of the benefits of using Eclipse

1. Eclipse is a free and open-source development model and so is constantly updated with community support.
2. Eclipse has large integration options with other tools.
3. Eclipse offers various perspectives for various needs, for example, Java Perspective for Java development, Debug Perspective for debugging purposes, etc.
4. It has built-in Source Control plugins that can connect to various source control tools like Git, Subversion, CVS, etc.
5. It offers various language-related tools such as compilation, execution, debugging, dependencies management, auto completion, etc.

10.7.2 Setting Up Eclipse

Eclipse works on top of your JDK. It offers the abstraction layer that we have defined already as one of the most important principles of OOP programming. It can only run properly when the IDE knows where to find the JDK. This means that once you have installed and opened the software, you should set up the installed JREs from the Preferences>Java>Installed JREs option.

You can simply select the JDKs that you want to use, while uncheck the ones that you do not want Eclipse to identify and implement in its Java abstraction scheme. To add a new JDK, simply identify the relevant directory through a browser window and the software will automatically ask you to confirm your selection if the JDK is found in the described file folder. Please visit Eclipse website to learn more about installation procedure: https://wiki.eclipse.org/Eclipse/Installation

10.7.2.1 Starting Eclipse

Java programmers must understand that Eclipse is much more than a simple IDE. It is a development atmosphere that contains the entire set of tools that you will ever need for developing Java projects. You can install the relevant Eclipse package from the website according to your particular computer system. Once you have installed the IDE, it is time to understand its basic setup.

The Eclipse development environment consists of four main sections. These include:

1. **Workspace:** The most important of these elements is the Eclipse workspace. It is the area that has a record of all your current projects.
2. **Perspectives:** The perspective is defined as a particular view of the projects that you have. A single perspective may allow a programmer to use multiple viewing options.
3. **Projects**: The projects contain your actual Java code and the details that you need for programming.

Views: Eclipse IDE provides various views which show project related metadata. For example, the Outline view shows various classes, variables, methods, etc. from the project (Figure 10.15).

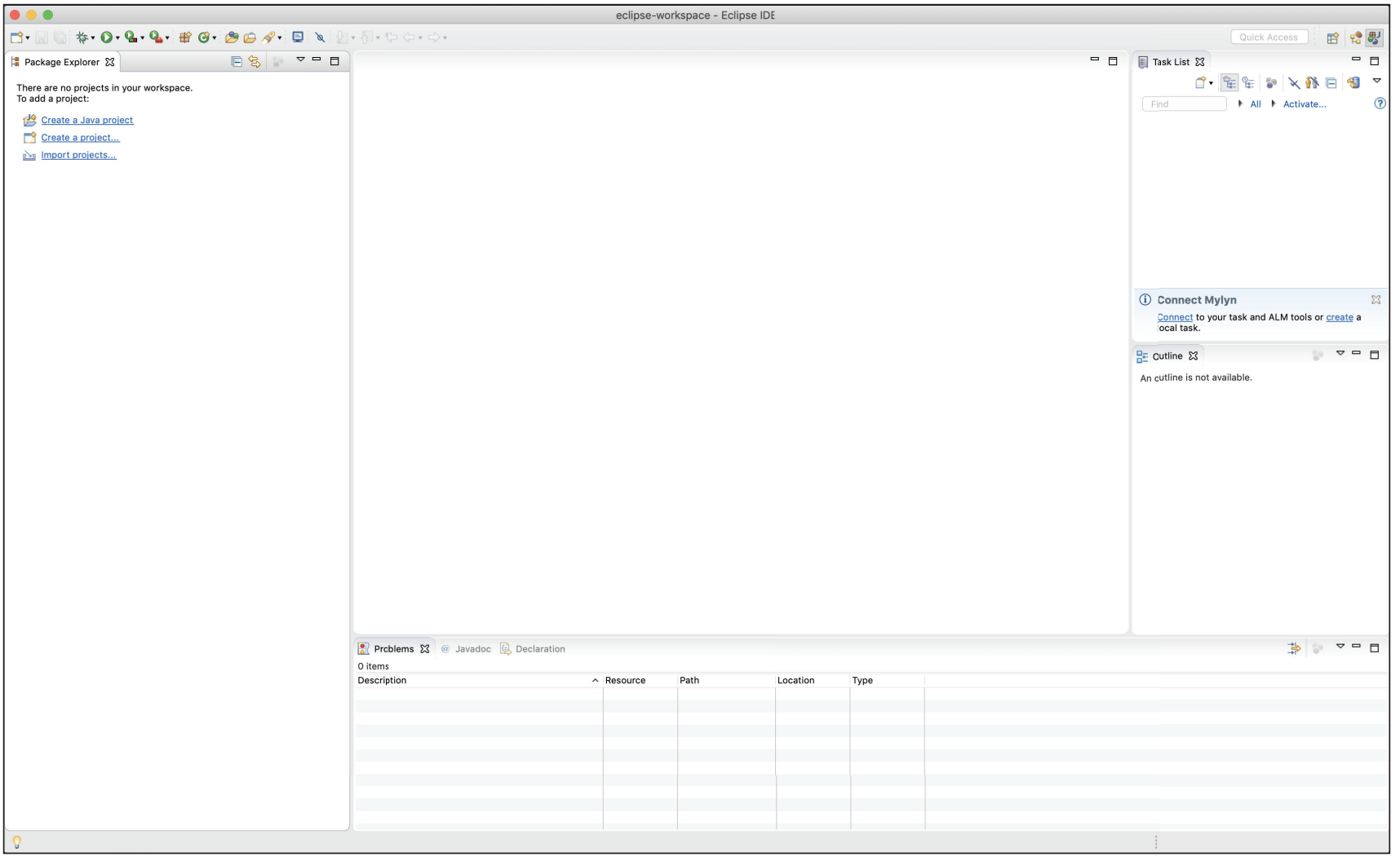

Figure 10.15 Eclipse IDE view.

The screenshot in Figure 10.15 is the example of the perspective that you get when you are viewing your projects in the workspace of an Eclipse window. The perspective provides access to the tools that you need to begin your Java projects. The singular tabs provide different views of the same perspective. The Outline and the Package Explorer are two important views that you will often use in Eclipse.

Remember, all views can be easily docked and moved around according to your convenience. You should take advantage of this flexibility and set up tools as you like them. However, we recommend that you start with the default settings that come with Eclipse.

10.7.2.2 Creating Projects

Creating a new project is easy in Eclipse. Follow Figure 10.16 to create a new project. You need to go to File menu, this will open up menu options. On that slide move your mouse to the New option which will open up a new option pane. On that option pane select Project. This will start a new project, where you can once again verify the specific JDK that you want to use and then click on the Finish button to end the wizard and start working on your project.

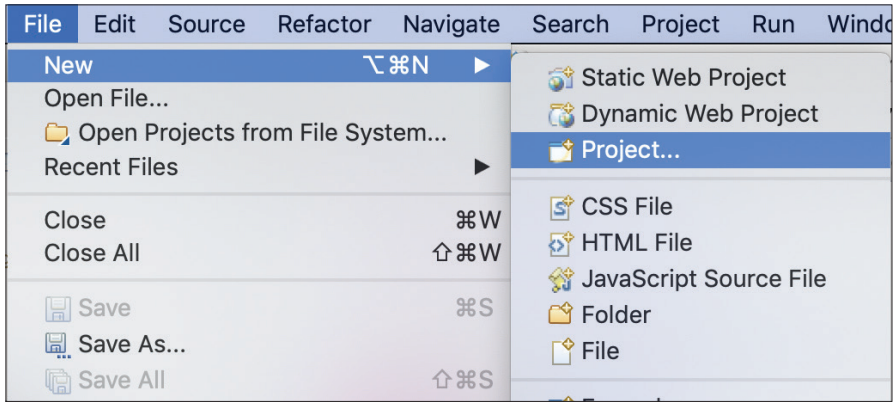

Figure 10.16 Create new project in Eclipse option.

A new Eclipse project automatically creates its source folder and prepares the development environment for immediate use. Understanding the principles of object-oriented programming is then the only thing required to create the necessary Java programs and code elements. Writing code is the next step and is accomplished using the Eclipse Package Explorer.

One of the main elements that we have already described above is the use of the `main()` method that you will see in every program, which can be used for testing as well as defining the required functionality in a Java program. The Logger class is also important when you are learning the use of Eclipse, as it provides information about how the application behaves during its execution, giving you information about how your code is performing.

10.7.2.3 Creating Packages

The Java Package Explorer is an excellent view which is available in the Java Perspective. Eclipse shows you all the information that you need for creating classes. However, you must create packages which are constructs that provide an address for the classes, where they can be mapped and saved according to the available file system on the computer.

The main idea of using Packages is to group the related classes like the way you place files in a folder. Think of a Package as a folder which can contain class files. Before you write your classes it is a good idea to design a good package structure to group the relevant classes. You can place all the relevant Java classes in one package like com.zonopact.mycoreclasses. With this structure, IDE will create three folders like "com" folder which contains another folder named "zonopact" and this "zonopact" folder contains "mycoreclasses" folder. Now let us look at how to create classes.

10.7.2.4 Defining Classes

Now that you have created a customized package and the source folder to store your classes, it is time to select a class from the New option in the File menu. This will open up the dialog box where you can name your class which will then appear on your

editing window in the Eclipse view. A good way to work on Eclipse, especially if you are a new user, is to close the views that you do not require so that you can easily work on your code. Eclipse remembers your choices and the next time you open the IDE, you have to deal with fewer windows.

Once you generate your class, Eclipse will create the shell for you and other package-related information at the top of your coding window. All you need to do is to now enter more details of your class, such as the variables and the methods that you want to be included as part of your class.

Once you start writing the code within the class, the Eclipse IDE can pick up all syntax errors and show them with underlines. This allows you to find that you have used an undefined word. It means that the final code copy that you create is free from syntax errors and will get compiled. However, the code can always have logical errors in it.

Now that you have initiated your Java class, it is time to prepare the coding elements. Remember, we already described that you must specify the access, the data type, and a unique variable name when defining the attributes of a Java class. The data type is of special importance in this context. It can either be of a primitive type such as int, Boolean, and long, or a non-primitive type such as String and Array, which you will learn in Chapter 11.

Next, you should set up the program logger, which we have described earlier when discussing the basic language tools.

Now we are back to coding the variables that show the attributes of the class. You can define any number of variables within the class easily in the Eclipse IDE. However, you do not have to manually type the getters and setters as you can go to the option of Source as shown in Figure 10.17, by moving the cursor to the class definite. When you select the Generate Getters and Setters button, it adds get and set methods for all the variables in the class. These are important methods to have in order to restrict the access to variables and reduce the exposure to maintain data integrity. Once you click this button, you get access to all the variables that are present under the class definition.

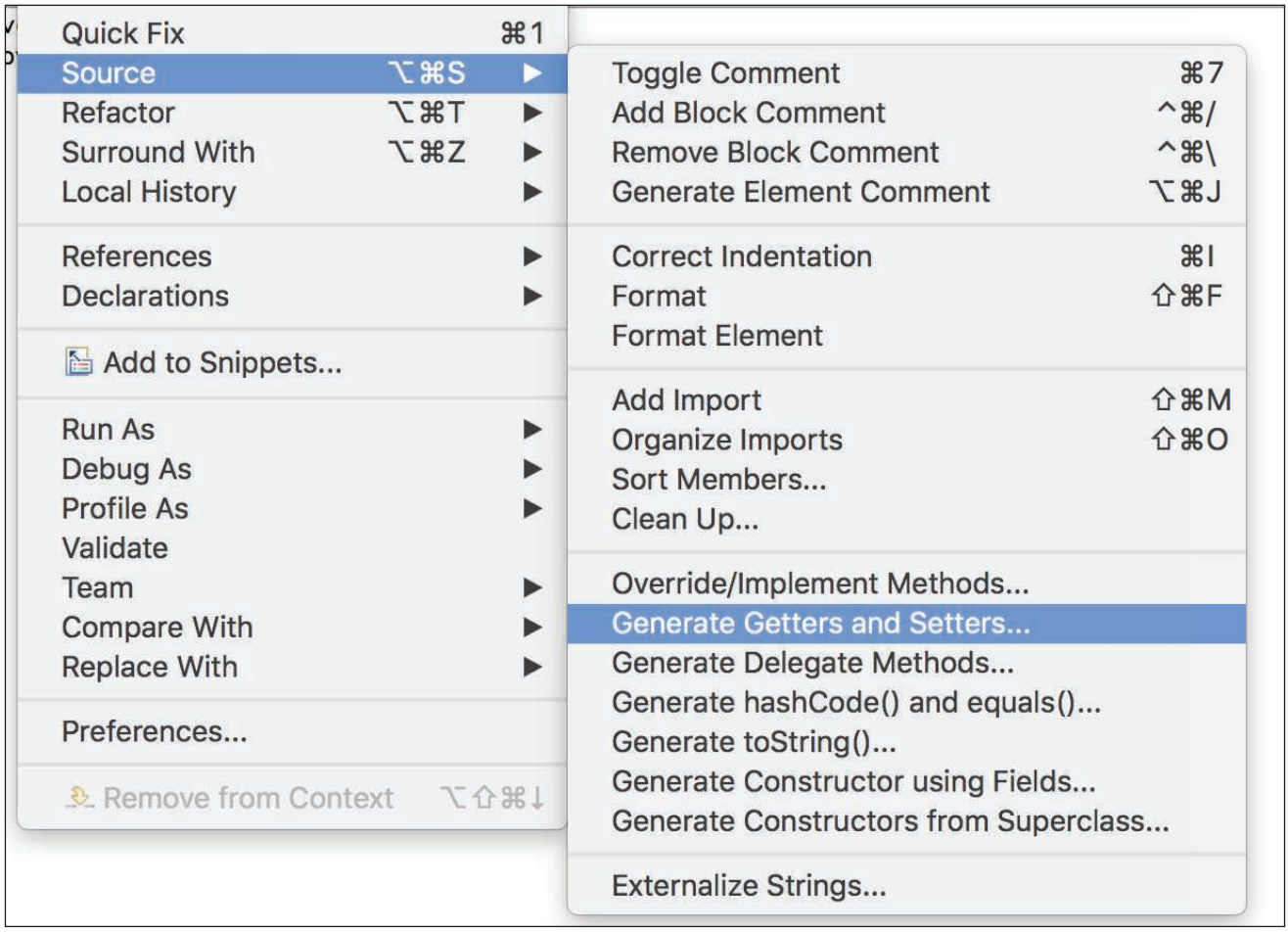

Figure 10.17 Option to Generate Getters and Setters.

An example of using the functionality can be best presented in the following manner:

```
public Book(String name, String genre, int pages, int width, int height, int length,
double cost)
{
    this.name = name;
    this.genre = genre;
    this.pages = pages;
    this.width = width;
    this.length = length;
    this.cost - cost;
}
```

This is a code that you will be able to generate with constructors once the getters and setters are already selected from our described class definition options.

10.7.2.5 Testing in Eclipse

Testing is important as it allows you to ensure that you have set up all the attributes in the right manner and you will not face any problems later on. The JUnit is a testing method that uses your mentioned constructors and then prints the total state of your defined object on the screen console. This allows you to find out that all the constructor instances are properly placed and they are working in the manner you want as a programmer.

Creating a test is easy in Eclipse, where you can simply right-click on your created class and then select the option of JUnit Test Case from New. This will open up the testing wizard of the IDE. Simply select the default options and go ahead by clicking on Finish, which will generate the JUnit test case.

As shown in Figure 10.18, running the test is easy, as you simply need to right click and go to "Run As" and select JUnit Test option. This will produce the status of all the variables that were described in the class using the Logger output.

Figure 10.18 JUnit test run option.

10.7.2.6 Java Class Behaviors

One advantage of employing Eclipse IDE is that it allows you to perform several access options directly. The access methods always follow a fixed pattern which is termed as the JavaBeans pattern and is built directly within the IDE. It includes declaring attributes as private, while the getters and setters are always public.

A getter does not take in parameters and returns the same data type as that of the attribute. The setter takes a single parameter which must be of same type as the attribute to which the parameter value is to set. The setter does not require to return a value.

Using behaviors also requires the calling of methods which is easy to perform in Java. The common way of invoking a method is simple where it follows the following scheme:

```
Object.requiredMethod();
```

The first part provides reference to the object that is used, while the "." defines that the next word to follow is the method to be employed. The method may take in arguments if required. The required parameters are all mentioned with the parentheses. Methods can be nested just like functions in C, where they can be used within each other for improved functionality. Consider the following example:

```
Logger l1 = Logger.getLogger(LearnJavaProgramming.class.getName());
l1.info("Name: " + p.getName());
```

This is a code where the `getName()` method is employed within the `getLogger()` method to create a nested method design. This functionality is commonly used when creating ordinary programs in Eclipse.

Summary

Java 13 and Java 11 brought some amazing improvements over the previous versions. Moreover, the major updates in Java 9 provide better heap management as well as the ability to reduce the stop-the-clock pauses that may appear during the execution of complex programs. This language is easy to employ if you already have some programming experience. However, if you are new to the language, we suggest that you first familiarize yourself with the principles of object-oriented programming rather than worrying about learning the required syntax and code elements.

In this chapter, we have learned the following concepts:

1. Basic principles of Java programming language.
2. Working of Java Virtual Machine and Compiler.
3. Garbage Collector and how it works.
4. Java Development Kit, Java Runtime Environment, and their difference.
5. What is Object and basic principles of Object-Oriented Programming.
6. Basic idea of Encapsulation, Inheritance, Polymorphism, and Abstraction.
7. How to write program in Java.
8. Basic syntax of Java language.
9. Reserved Words, Variables, and Constructor.
10. New features in Java 9.
11. How to set up Eclipse IDE and use it to write programs.

With this you got the information that you require in order to start programming with Java. We strongly encourage that you experiment with the language to create unique programs that allow you to generate the required functionality. With the use of Eclipse IDE, you will find coding in Java to be an easy affair, where you can gradually move on to create more complex programs as you learn about the various aspects of this excellent programming platform.

In Chapter 11, you will learn the structure of a Java program and also explore concepts such as identifiers, keywords, literals, variables, code blocks, comments, Java packages, primitive classes, static and non-static methods, string options, arrays, enums, overriding, autoboxing, unboxing, various types of operators, expressions, control flow, loops, etc.

Multiple-Choice Questions

1. Which of the following tools is utilized for compiling Java code?
   - **(a)** Jar
   - **(b)** Java
   - **(c)** Javadoc
   - **(d)** Javac
2. Which of the following tools is used to execute Java code?
   - **(a)** Javac
   - **(b)** Javadoc
   - **(c)** rmic
   - **(d)** Java
3. Which tool is available to generate API documentation in HTML format from Java source code?
   - **(a)** Javadoc
   - **(b)** Javamanual
   - **(c)** Javahelp
   - **(d)** None of the above

4. What is the full form of jar?
   (a) Java Archive
   (b) Java Archive Runner
   (c) Java Application Runner
   (d) None of these

5. Which of the following is not known as a keyword in Java?
   (a) Assert
   (b) Boolean
   (c) Abstract
   (d) Finalize

Review Questions

1. What is the difference between JDK and JRE?
2. How is JVM useful in executing Java code?
3. What is Garbage Collection and how does it work?
4. How do you define Object and Class?
5. What is Object-Oriented Programming?
6. List Object-Oriented Programming principals.
7. How is inheritance useful?
8. What is package and how do you create one?
9. What is access modifier? Name and describe all of them.
10. What is the default access modifier?
11. What is reserved word in Java? Name at least 10 and explain in detail.
12. What is the difference between constructor and method?
13. What is the use of constructor?
14. Can method be private?
15. What is variable? Give some examples.
16. What are some of the prominent features of Java 9?
17. How can one set up Eclipse and create a new project?
18. Can you test a project in Eclipse?

Exercises

1. Think of a real-life example of inheritance, polymorphism, abstraction, and encapsulation. Design class structure for these examples.
2. Draw a flow chart to explain the flow of Java Code from plain text to executable.
3. Create a chart to explain the benefits of using Java 9 over Java 8.
4. Using Eclipse IDE, create a simple project, add one package and 3 classes.

Project Idea

Take an example of a vending machine that dispenses food items upon accepting money. State the object-oriented principal that this vending machine is based on and explain the entities and their relationships.

Recommended Readings

1. Oracle Java Doc: https://docs.oracle.com/javase/tutorial/
2. W3schools: https://www.w3schools.com/java/